VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

AUTOMATIZOVANÉ TESTY HARDWARU S VYUŽITÍM FRAMEWORKU PYTEST

AUTOMATED HARDWARE TESTS USING THE PYTEST FRAMEWORK

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **LUKÁŠ TESAŘ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JIŘÍ MATOUŠEK, Ph.D.**
SUPERVISOR

BRNO 2025

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

158418

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Tesař Lukáš**
Program:	Informační technologie
Název:	**Automatizované testy hardwaru s využitím frameworku pytest**
Kategorie:	Návrh číslicových systémů
Akademický rok:	2024/25

Zadání:

1. Seznamte se s prostředím využívaným sdružením CESNET pro automatizované testování síťových karet s FPGA čipem.
2. Nastudujte možnosti frameworku pytest určeného pro implementaci automatizovaných testů.
3. Navrhněte po konzultaci s vedoucím práce a odborným konzultantem automatizované testy pro alespoň dvě skupiny funkcí síťových karet s FPGA čipem. Vyberte si takové skupiny funkcí, pro které prostředí, využívané sdružením CESNET, prozatím žádné testy neobsahuje.
4. Implementujte nově navržené automatizované testy.
5. Ověřte funkčnost implementovaných testů nad zvolenou aplikací pro síťovou kartu s FPGA.
6. Zhodnoťte dosažené výsledky práce a diskutujte možnosti jejího dalšího pokračování.

Literatura:

- Dle pokynů vedoucího práce.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

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

Vedoucí práce:	**Matoušek Jiří, Ing., Ph.D.**
Konzultant:	Ing. David Vodák
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1.11.2024
Termín pro odevzdání:	14.5.2025
Datum schválení:	31.10.2024

Abstrakt

Sdružení CESNET vyvíjí pro své potřeby i komerční partnery řadu vysokorychlostních síťových karet postavených na technologii FPGA. Tato práce se zaměřuje na jejich testování. V rámci dané problematiky jsou popsány samotné síťové karty a také stávající testovací prostředí, které sdružení CESNET k jejich testování využívá. Následně jsou navrženy dvě sady automatizovaných testů: první je zaměřena na komponentu MAC, přítomnou ve většině karet sdružení a druhá je více specifičtější, vyvinutá pro testování funkcionality MVB paketového filtru v aplikaci NIC. Tyto sady jsou následně také implementovány a po ověření funkčnosti integrovány do prostředí pro testování síťových karet na serverech CESNET. V rámci implementovaného řešení je kladen důraz na automatizaci a oddělení jednotlivých testovaných případů.

Abstract

The CESNET Association develops a range of high-speed network cards based on FPGA technology for its commercial partners and also for its own needs. This thesis focuses on automated testing of such cards. The network cards themselves and the existing testing environment used by CESNET for their testing are described within the scope of the topic. Subsequently, two sets of automated tests are proposed, with the first one oriented on the MAC filter present in most of CESNET's cards and the other, which is focused on the functionality of the MVB packet filter in the NIC application. These sets are then implemented, verified for functionality, and integrated into the testing environment for network cards on CESNET servers. The implemented solution emphasizes automation and the separation of individual test cases.

Klíčová slova

síťové karty, testování, automatizované testy, hardware, Python, pytest, CESNET, NFB, NDK, NIC

Keywords

network cards, testing, automated tests, hardware, Python, pytest, CESNET, NFB, NDK, NIC

Citace

TESAŘ, Lukáš. *Automatizované testy hardwaru s využitím frameworku pytest.* Brno, 2025. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Matoušek, Ph.D.

Automatizované testy hardwaru s využitím frameworku pytest

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Matouška, Ph.D., pod záštitou sdružení CESNET. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Lukáš Tesař
13. května 2025

Poděkování

Především bych rád poděkoval panu Ing. Davidu Vodákovi ze sdružení CESNET za vedení praktické části bakalářské práce a cenné rady při jejím vypracovávání. Dále mé poděkování patří vedoucímu mé práce, panu Ing. Jiřímu Matouškovi, Ph.D. za odbornou pomoc a vedení při psaní technické zprávy.

Obsah

Kapitola 1

Úvod

V dnešní době, kdy je Internet neodmyslitelnou součástí našich životů a na jeho existenci stojí celá řada i kriticky důležitých aplikací, vzrůstá potřeba mít pod kontrolou síťový provoz, a to zejména pro provozovatele datacenter a peeringových center, které dostupnost mnohých aplikací zajišťují. Neziskové sdružení CESNET, pro nějž a pod jeho záštitou byla tato práce implementována, má dlouholetou zkušenost s provozováním páteřních sítí [7]. Potřeba kontroly síťového provozu pramení zejména ze zvyšujícího se množství dat, které danými sítěmi proteče a které musí jednotlivé uzly zpracovat, ale také ze stále častějších hrozeb, které jsou neustále inovovány, a tedy i boj s nimi musí být neustále inovován. Mezi takové hrozby patří pokusy o vniknutí do systémů a také DDoS[1] útoky.

Mnohé tyto problémy se dají řešit správným monitorováním a detekcí hrozeb. K tomu je u větší infrastruktury zapotřebí hardwaru, který je schopný zpracovat velké množství síťového provozu. Sdružení CESNET pro tyto účely vyvíjí vlastní řešení, zejména síťové karty s propustností až 400 Gb/s [3]. Využito je přitom technologie FPGA[2].

Vývoj těchto karet i softwaru obecně je náročná disciplína a kromě zkušených odborníků vyžaduje dobrou testovací infrastrukturu. Ta dokáže odhalit mnoho chyb, které by jinak mohly zůstat skryté a způsobit problémy později. Tato testovací infrastruktura by měla být pokud možno co nejvíce automatizovaná, aby se dala použít pro odhalování jak stávajících, tak i nově vzniklých chyb při vývoji. Právě problematika testování je stěžejním tématem této práce.

Výstupem je implementace dvou sad funkcionálních testů pro dvě různé části síťových karet sdružení. Ty jsou navrženy s ohledem na stávající řešení a integrovány přímo do něj.

V druhé kapitole mé práce je představena architektura síťových karet sdružení CESNET, nástroje pro práci s nimi a testovací prostředí pro ně založené na testovacím frameworku `pytest`. Třetí kapitola se zabývá představením samotného frameworku `pytest`, který je použit pro implementaci nových testovacích sad. Čtvrtá kapitola pojednává o jejich návrhu a implementaci.

V páté kapitole je rozebráno ověření funkčnosti implementovaných testů včetně popisu jejich integrace. V poslední kapitole jsou shrnuty výsledky mé práce, zejména pak odhalené chyby a možnosti dalších rozšíření a vylepšení.

[1] Distributed Denial of Service — typ útoku, při kterém je oběť zahlcena typem či objemem provozu, který není schopna odbavit

[2] Programovatelná hradlová pole — technologie pro dynamický vývoj hardware pomocí programovacího jazyka — například VHDL.

Kapitola 2

Síťové karty sdružení CESNET

Předtím, než bude vysvětlena problematika síťových karet, je dobré zmínit, k jakému účelu jsou síťové karty ve sdružení CESNET vlastně vyvíjeny. Tímto primárním cílem je usnadnit řešení některých problémů, které jsou spojeny s provozováním počítačových sítí.

2.1 Problémy s provozováním počítačových sítí

Tato kapitola vychází z [19, str. 4].

Při provozování počítačových sítí je důležité zejména zajistit dostupnost sítě jako celku a ochránit ji před neoprávněným přístupem či útoky na zdroje. Proti většině těchto hrozeb se lze účinně bránit automatickou detekcí a následně filtrováním škodlivého provozu. Pro automatickou detekci je nutné mít přehled o tom, jaký provoz se v síti vyskytuje (tzv. viditelnost síťového provozu).

2.1.1 Správa sítí

Problém správy sítí lze rozdělit na tři hlavní podčásti:

1. **Sběr dat** (monitorování), kdy je stěžejní získat co největší množství pokud možno relevantních dat,
2. **Zpracování dat**, kdy se data získaná z předchozího kroku filtrují, agregují, případně se sdružují do celků reprezentujících jednu událost
3. **Prezentace dat a odezva**, bez které by monitorování nemělo smysl. V tomto kroce se vybraná data mohou přehledně zobrazovat např. grafickou formou. Také je zde důležitá případná reakce na neobvyklé události (zvýšené zatížení sítě, podezřelý provoz apod.)

Pro účel této práce je relevantní zejména krok 1, který je dále rozebrán podrobněji.

2.1.2 Monitorování síťového provozu

Velice důležitá je schopnost efektivně monitorovat příchozí provoz a získávat tak cenné informace o datových tocích, počtech přijatých a odeslaných dat a dalších informacích. V běžných počítačových sítích se tato řeší pomocí protokolů jako je SNMP nebo NetFlow od firmy Cisco [19]. V páteřních sítích, které běžně mají propustnost stovek Gb/s [7],

popřípadě ve velkých datacentrech je vhodnější použití specializovaného hardware, jako jsou právě akcelerační síťové karty sdružení CESNET. Tyto karty ale následně mohou výše zmíněné protokoly využívat také, viz nástroj `ipfixprobe` [35].

2.2 Projekt Liberouter

Vývojem akceleračních síťových karet se zabývá projekt Liberouter, který je výsledkem spolupráce českých univerzit[1] a sdružení CESNET [13]. V rámci projektu vznikla již řada vědeckých publikací[2], prototypů i reálně prakticky využitelných řešení. Příkladem může být síťová karta schopná zpracovat síťový provoz o rychlosti 400 Gbps [3].

Pro vývoj těchto zařízení je využita technologie FPGA. Ta umožňuje prototypovat a vyvíjet specializovaný hardware bez nutnosti mít jej vyroben přímo na míru. Tyto síťové karty — označované jako NFB[3] — jsou typicky zařízení s rozhraním PCI-Express, vyvinuté a stavěné na zpracování ethernetového provozu [6].

2.3 Architektura síťových karet

Aby byla technologie co nejvíce použitelná a škálovatelná v praxi, sdružení CESNET má velkou část funkcionality naimplementovánu pomocí vlastního frameworku nazvaného NDK[4]. Tento framework je navržen s ohledem na rychlý a efektivní vývoj síťových zařízení akcelerovaných právě pomocí FPGA. Framework obsahuje množství znovupoužitelných modulů, které lze využít pro sestavení konkrétní aplikace. Součástí je veřejná dokumentace a referenční řešení *Minimal NDK application*, které v zásadě neimplementuje žádnou vnitřní fukcionalitu a nijak nezpracovává pakety [4].

NDK využívá sběrnice nazvané **MFB** (*Multi-Frame Bus*) a **MVB** (*Multi-Value Bus*), tedy je schopný přijímat, zpracovávat a odesílat paralelně více paketů najednou v jednom hodinovém cyklu. Díky tomu je, co se týče propustnosti, škálovatelný od desítek do stovek gigabitů za sekundu [16].

Na obrázku 2.1 lze vidět nejdůležitější komponenty síťové karty s NDK frameworkem. Samotná aplikace je připojena k síťovému modulu a DMA modulu. DMA modul může obsluhovat (a typicky obsluhuje) více kanálů. Z obrázku jsou vynechány některé podrobnosti jako připojení k operační paměti, kontrolní sběrnici spojující všechny moduly a sběrnici pro synchronizaci časové značky generované jednotkou TSU[5].

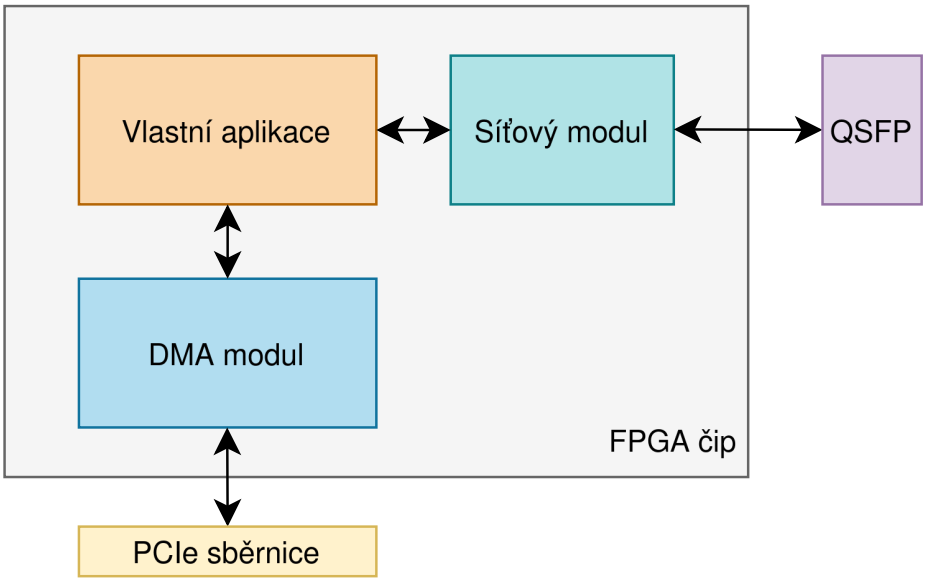
[1] VUT, MUNI, ČVUT a VŠE

[2] https://www.liberouter.org/publications/

[3] Network FPGA Board — obecné označení hardwarově akcelerovaných síťových karet s FPGA čipem vyvíjených sdružením CESNET

[4] Network development kit

[5] Time Stamp Unit



Obrázek 2.1: Vysokoúrovňový pohled na kartu NDK (vychází z [5])

Síťový modul obsluhuje příjem a odesílání ethernetových rámců z/do sítě. K tomu používá dva porty: jeden určený pro příjem — RX a druhý pro vysílání — TX. **DMA modul** se stará o vysokorychlostní přenos dat k hostujícímu PC [5].

2.3.1 DMA

Direct memory access — DMA — je technika, která umožňuje komunikaci s periferními zařízeními — a mezi hardwarem obecně — bez intervence, nebo jen s minimální intervencí CPU. Díky tomu lze dosáhnout vysoké propustnosti. Dnes je využívána ve většině desktopových i serverových řešení pro komunikaci s většinou periferií [2].

Při použití DMA jsou jednotlivé komponenty schopny přistupovat do operační paměti napřímo. Tento přístup je buď pouze pro čtení, pouze pro zápis nebo může být obousměrný[6].

Síťové karty sdružení CESNET využívají techniku DMA prostřednictvím vlastního řadiče *DMA Medusa*. Ten je navržen tak, aby byl škálovatelný až do propustnosti 400 Gb/s, zatímco jiná existující řešení jsou schopna dosáhnout propustnosti jen okolo 100 Gb/s [12].

DMA je v NDK členěno na tzv. *DMA streamy* (též DMA porty), které přenašejí pakety z/do karty a jejichž počet typicky koresponduje s počtem ethernetových portů (RX a TX). Ty jsou dále děleny na *DMA kanály*, nezávislé fronty, do kterých lze data posílat nebo z nich naopak data přijímat [4].

DMA Medusa je proprietární řešení sdružení CESNET a jako takové není zahrnuto v otevřených zdrojových kódech frameworku NDK na platformě GitHub [16].

2.3.2 NDK aplikace

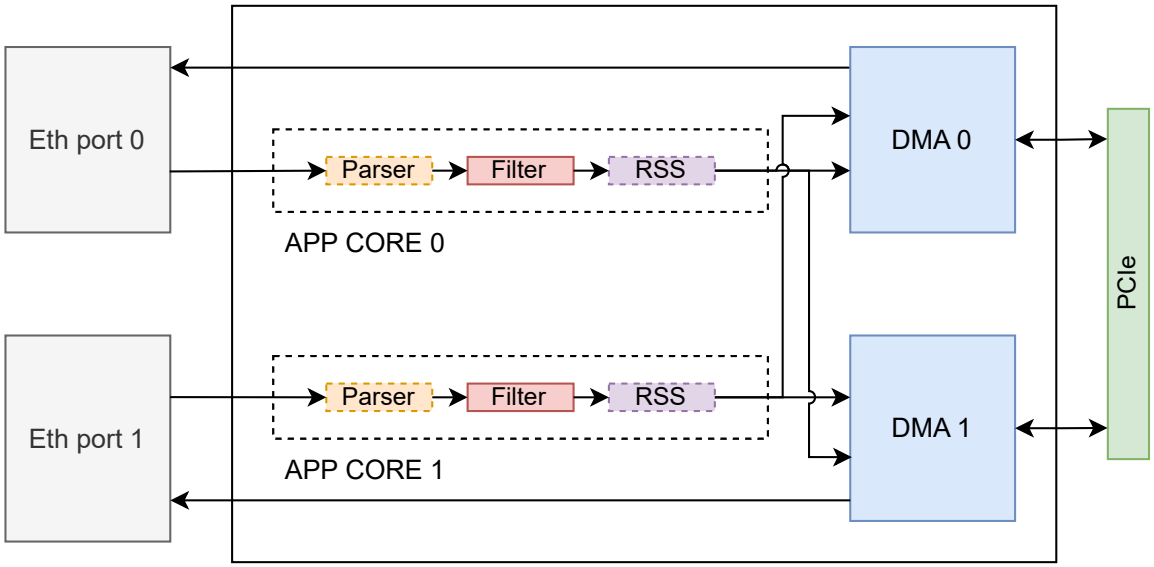
Z obrázku 2.1 je patrné, že konkrétní design pro síťovou kartu vybudovaný s pomocí NDK zpravidla obsahuje uživatelskou aplikaci, která může být vyvinuta k libovolnému účelu — např. právě k monitorování. V této práci je jedna z testovacích sad zaměřena na jeden konkrétní design nazvaný *NIC*.

[6] tzv. dual-ended DMA

2.4 Aplikace NIC

NDK-APP-NIC (též aplikace NIC) je aplikace pro FPGA síťové karty postavená na frameworku NDK, která má za cíl vytvořit tzv. *SmartNIC* s podporou *DPDK*[7] [15]. Termínem **SmartNIC** se označuje síťová karta s určitou úrovní programovatelnosti, jako například konfigurovatelný filtr paketů. Pro tato zařízení se také používají termíny DPU[8] či IPU[9] [18].

DPDK je sada knihoven a ovladačů s přímou podporou v linuxovém jádře, určená pro rychlé zpracování paketů. Výhodou NDK-APP-NIC je dále přímá podpora frameworku NFB (podrobněji popsaného v sekci 2.6).



Obrázek 2.2: Tok paketu v aplikaci NIC (vychází z [15, Firmware Overview])

Diagram 2.2 znázorňuje tok paketů jedním i druhým směrem od PCIe portu k ethernetovým portům. Mnoho FPGA aplikací podporuje více aplikačních jader a nejinak je tomu i u designu NIC, kde konkrétně na tomto diagramu jsou znázorněna dvě aplikační jádra připojená ke 2 ethernetovým portům, obecně jich však může být více [15]. Každé aplikační jádro obsahuje několik prvků, z nichž každý implementuje jinou funkcionalitu.

Mezi klíčové komponenty designu NIC patří:

1. **parser** schopný pracovat na rychlosti linky, označený žlutým obdélníkem v diagramu 2.2,
2. obecný MVB **paketový filtr** s *obsahově adresovatelnou pamětí (TCAM)*, přítomný mezi parserem a RSS komponentou a zvýrazněný červenou barvou,
3. distribuce paketů mezi jednotlivé DMA kanály (fronty) pomocí **komponenty RSS**, která využívá hashů, na obrázku 2.2 označenou fialově.

Z obrázku byla dále vynechána komponenta **CTT** (*Connection Tracking Table*), protože není žádným způsobem relevantní k obsahu této práce. Tato komponenta je ve výchozím stavu **vypnutá**.

[7] Data Plane Development Kit
[8] Data Processing Unit
[9] Infrastructure Processing Unit

Komponenta **RSS** umožňuje distribuci paketů i do jiných DMA portů, než které náleží danému jádru, což je v diagramu znázorněno šipkami vedoucími z aplikačních jader do obou DMA komponent.

2.5 Komponenta MAC

V rámci síťového modulu se dále nachází komponenta MAC (dále též MAC filtr), která je především zodpovědná za filtraci paketů na 2. vrstvě síťového modelu TCP/IP. Ta je rozdělena na LLC podvrstvu (IEEE 802.2) a MAC podvrstvu, kterou zahrnuje zejména protokol Ethernet (802.3). Pakety 2. síťové vrstvy se zpravidla také označují pojmem *rámce*. K adresaci obou stran komunikace je využito tzv. *MAC adresy*, což je 6-bajtový, zpravidla celosvětově unikátní identifikátor. Dále protokol definuje také mechanismy opravy chyb při přenosu a využívá kontrolního součtu (FCS — Frame Check Sequence) [8].

Komponenta má několik režimů:

1. `normal` – standardní režim, při kterém jsou zpracovány pouze pakety, jejichž cílová MAC adresa se shoduje s některou z MAC adres nastavených v tabulce,
2. `promiscuous` – promiskuitní režim, který umožní průtok veškerého provozu,
3. `multicast` – povolí pouze MAC adresy v tabulce a adresy spadající do multicastových skupin,
4. `broadcast` – povolí pouze MAC adresy v tabulce a broadcast adresy.

2.6 NFB framework

Následující sekce vychází z [6].

Pro komunikaci s kartami postavenými na NDK a jejich řízení je využit vlastní framework NFB. Ten obsahuje několik částí:

1. **ovladač** pro linuxové jádro,
2. userspace **knihovny** `libnfb` a `pynfb` využitelné v aplikacích a
3. **nástroje příkazové řádky**, na nichž je z velké částí postaveno testovací prostředí pro karty.

Implementační část této práce je postavená především na částech 2 a 3, proto budou dále rozebrány podrobněji.

2.6.1 Knihovna `libnfb`

Pro nízkoúrovňovou práci s kartami sdružení CESNET slouží knihovna `libnfb`. Ta poskytuje C/C++ API pro základní operace, jako je vysokorychlostní odesílání/přijem paketů a zápis do registrů zařízení. Dále poskytuje funkce pro nahrávání firmwaru a práci s Device-tree, ty jsou ale určeny jako interní API pro nástroje příkazové řádky

Pro tuto práci je podstatná jen podmnožina funkcí z těch, které knihovna exportuje[10]. První z nich je `nfb_open(path)`, sloužící k otevření souboru se zařízením karty ve virtuálním souborovém systému, kam jej exportuje ovladač. Funkce vrací ukazatel na strukturu

[10]Export — veřejné zpřístupnění symbolu z modulu či knihovny pro využití jinými moduly

`nfb_device`, která je poté předávána dalším funkcím při následné práci s kartou. Komplementární funkcí je `nfb_close`, která takto otevřené zařízení korektně uzavře.

Dále je součástí této podmnožiny sada funkcí uvozených prefixem `ndp_`, které slouží k manipulaci s frontami a následně odesílání či příjmu paketů:

- `ndp_open_rx_queue(device, queue_id)` – slouží pro otevření fronty s indexem `queue_id`, vrací handle dané fronty
- `ndp_queue_start(queue)` – otevře frontu pro přenos — příjem nebo vysílání paketů
- `ndp_rx_burst_get(queue, pkts, count)` – přečte z fronty shluk paketů, vrací počet přečtených
- `ndp_rx_burst_put(queue)` – vrátí pakety zpět do fronty – nutné v určitém čase zavolat pro všechny pakety přečtené funkcí `ndp_rx_burst_get`
- `ndp_close_rx_queue(queue)` – zavře dříve otevřenou frontu

Výše uvedených funkcí je využito v implementační částí této práce. Obdobné varianty funkcí s infixem `_tx_` existují i pro TX fronty.

2.6.2 Nástroje příkazové řádky

Balík `nfb-framework` obsahuje po své instalaci do systému množinu nástrojů spustitelných v příkazové řádce. Tyto příkazy se dělí na:

1. **konfigurační** příkazy a příkazy pro sběr dat, začínající prefixem `nfb-` a
2. **příkazy pro datové přenosy**, zejména odesílání a přijímání paketů, uvozené prefixem `ndp-`.

Příkazy mají některé společné přepínače (zmíněny jsou jen podstatné):

- `-d` – Specifikace zařízení, které bude použito. K danému serveru může být takových zařízení připojeno i více. Může být specifikováno plnou cestou k souboru zařízení (např. `/dev/by-pci-slot/0000:3b:00.0`), jeho pořadím v rámci sběrnice nebo některým ze symbolických odkazů v rámci adresáře `/dev`.
- `-i` – Výčet rozhraní (DMA kanálů), která mají být použita.
- `-h` – Výpis nápovědy. K příkazům nejsou dodány manuálové stránky, nápověda slouží jako jejich náhrada.

`nfb-boot`

Prvním z příkazů, který se používá v typickém scénáři pro práci s kartou, je `nfb-boot`, který slouží k nahrání firmware do karty. Ta má typicky dva paměťové sloty pro uložení firmware: *konfigurační* (0) a *recovery* (1). Při studeném startu se načte firmware z recovery slotu, ten však většinou neposkytuje potřebnou funkcionalitu a slouží jako stub[11] pro nahrání jiného firmwaru. Většinou je tedy zapotřebí nahrát vlastní firmware do slotu určeného pro běžný provoz:

[11] *Stub* (česky pahýl) je označení pro software či funkci, která slouží jako dočasná, případně jednodušší náhrada za jeho plnohodnotnou verzi, poskytující podmnožinu funkcí dostatečnou pro testování či konkrétní případ použití.

```
nfb-boot -f 0 <firmware>.nfw
```

kde přepínač `-f` nahraje firmware do slotu `0` ze souboru s příponou `.nfw` a spustí kartu s tímto nově nahraným firmware. Tento soubor je pouze komprimovaný archiv obsahující Device tree ve zdrojové a binární podobě a *bitstream*[12] syntetizovaného designu. Mezi další přepínače patří `-F`, sloužící pouze pro nahrání firmware a `-b`, který porovnáním otisku obrazů ověří, že se již shodný firmware v kartě nenachází a poté případně zařídí jeho nahrání [6, NFB tools — nfb-boot].

`nfb-info`

Dalším z rodiny konfiguračních příkazů je `nfb-info`, vypisující informace o zařízení. Mezi tyto informace patří:

- Název zařízení
- Sériové číslo
- Počet síťových rozhraní (ethernetových portů)
- Projektový název a verze nahraného firmwaru
- Informace o připojení: PCI adresa a numa node[13], ke kterému je zařízení připojeno

`nfb-eth`

Pro konfiguraci síťových rozhraní a získávání statistik slouží příkaz `nfb-eth`. Ten má několik důležitých přepínačů:

- `-e` – povolení/zakázání ethernetového rozhraní (0 — zakázáno, 1 — povoleno)
- `-P` – nastavení související s PMA
  - `-c` – nastavení PMA módu či povolení (`+`), resp. zakázání (`-`) vlastnosti
- `-M příkaz` – nastavení RXMAC filtru – zde je možné přidávat/odebírat MAC adresy z tabulky a nastavit promiskuitní/broadcast/multicast mód
- `-l`, `-L` – přepínače pro manipulaci s minimum frame length/maximum frame length (MTU)

Spuštěním bez parametrů jsou vypsány základní čítače RX i TX rozhraní. Ty ale zpravidla fungují až po povolení ethernetu příkazem `nfb-eth -e 1`, což je nutné téměř vždy, protože aplikační jádro se zpravidla zasekne, pokud TX rozhraní nepřijímá pakety [6, NFB tools — nfb-eth].

[12] Bitstream (česky *proud bitů* nebo *bitový tok*) označuje sekvenci bitů přenášenou nebo ukládanou jako souvislý proud dat, obvykle bez pevné struktury na vyšší úrovni (např. bez zřetelného dělení na slova nebo rámce); používá se např. při sériovém přenosu dat, enkódování multimediálního obsahu nebo právě konfiguraci FPGA.)

[13] NUMA — non-uniform memory access — v serverových řešeních typicky bývá více CPU (každé s vlastní pamětí), kdy je poté celek označován jako NUMA systém a jednotlivé CPU celky jako NUMA nodes [30]

Pro účel této práce je relevantní také výše zmíněný přepínač `-P` spolu s přepínačem `-c`. Ty umožňují povolení tzv. *lokálního PMA loopbacku*[14], který je pro testování vhodný. To se provádí příkazem `nfb-eth -Pc "+PMA local loopback"`.

Dále je podstatný přepínač `-S`, který vypisuje více statistik ethernetových rozhraní. V průběhu implementace práce byl (nikoliv však autorem) nástroj rozšířen o přepínač `-j`, který tato data poskytne ve strojově zpracovatelném formátu JSON.

`nfb-dma`

Příkaz `nfb-dma` slouží k zobrazení **statistik DMA** kanálů[15] v obou směrech (*RX* i *TX*). U každého z kanálů vypíše počet zpracovaných (`Received`, resp. `Sent`) paketů s počty zpracovaných bajtů a u RX DMA zobrazí počet zahozených (`Discarded`) paketů a bajtů. Spuštění bez parametrů vyvolá výpis statistik všech front, toto chování lze dále omezit na RX (přepínačem `-r`) nebo TX (přepínačem `-r`). Velice vhodné je použití přepínače `-T`, který poskytne souhrnné statistiky (součet nenulových hodnot u všech front):

```
$ nfb-dma -r
------------------------------ RX00 NDP controller ----
Received : 0
Received bytes : 0
Discarded : 5784
Discarded bytes : 8936280
------------------------------ RX01 NDP controller ----
...
$ nfb-dma -rT
------------------------------ RX SUM -----------------
Received : 0
Received bytes : 0
Discarded : 5784
Discarded bytes : 8936280
```

Tohoto výpisu je poté využito v první testovací sadě, kde je kontrolováno propuštění, resp. zahození paketů dle pravidel filtru.

`ndp-trasmit` a `ndp-receive`

Poslední dva příkazy slouží k odesílání, resp. příjmu paketů do/ze síťové karty. Kromě společných přepínačů zmíněných výše lze u těchto použít přepínač `-f` s cestou k souboru PCAP, obsahujícím pakety ve formátu používaném knihovnou `libpcap` [10]. `ndp-transmit` obsah tohoto souboru odešle do zařízení, `ndp-receive` po spuštění začne naslouchat na specifikovaných rozhraních a soubor naplní případnými přijatými pakety.

2.6.3 Jazyk JSON

Tato kapitola vychází z [1].

JSON (JavaScript Object Notation) je strukturovaný jazyk pro přenos dat, který získal popularitu zejména díky své jednoduché syntaxi a poměrně snadné čitelnosti jak pro stroj, tak pro člověka, i když existují lidsky čitelnější jazyky, např. YAML (2.7.1). Ačkoliv je

[14] Loopback je speciální mód přenosového zařízení, kdy je přijmutá sekvence dat odeslána zpět na výstup.
[15] v tomto kontextu také front

syntaxe JSONu odvozena z jazyka ECMAScript, formát samotný je jazykově nezávislý a podporován napříč většinou moderních programovacích jazyků.

JSON byl standardizován v roce 2006 jako RFC 4627. Od té doby prošel mírným vývojem, s aktuální verzí v RFC 8259. Jazyk má celkem 6 základních typů — 4 skalární (*řetězce*, *čísla*, *booleovské hodnoty* a hodnotu *null*) a dva strukturované datové typy (*objekty* a *pole*). JSON data, která mají být přenositelná mezi systémy, *musí* využívat kódování UTF-8.

Objekt je struktura reprezentovaná párem složených závorek (`{}`) obalujících nula nebo více párů typu jméno/hodnota (dále také jako člen). Jméno je řetězec unikátní v rámci objektu, jinak je chování zpracovávajícího software nedefinováno. **Pole** je oproti objektu definováno dvojící hranatých závorek (`[]`) obalujících nula nebo více hodnot oddělených čárkami. JSON ve své základní variantě nepodporuje komentáře.

Jak již bylo zmíněno, nástroj `nfb-eth` využívá jazyk JSON pro výpis statistik ve strojově zpracovatelném formátu. Tyto statistiky jsou do velké míry kompatibilní se standardem RFC 1271 [32], v některých případech však redefinuje význam jednotlivých klíčů [36].

2.6.4 RFC 1271

Remote Network Monitoring MIB (zkráceně **RNM MIB**) je rozšíření databáze MIB[16] o statistiky zaměřené na vzdálený monitoring sítí prostřednictvím sond [32, kapitola 4]. Ta souvisí s dříve zmíněným protokolem SNMP[17], využitelným pro monitorování sítí nebo síťových prvků.

SNMP definuje pojem monitorované objekty, které jsou popsány pomocí jazyka SMI[18]. Tyto objekty jsou adresovány pomocí *OID* (**O**bject **ID**entifier) — unikátního identifikátoru, který jednoznačně určuje každý objekt v rámci hierarchie. OID tvoří stromovou strukturu, kde jednotlivé objekty představují listy, zatímco skupiny objektů jsou reprezentovány nelistovými uzly. Tato hierarchie umožňuje efektivní organizaci a přístup k monitorovaným datům. MIB je poté databáze těchto monitorovaných objektů, která slouží jako standardizovaný prostředek pro výměnu informací mezi spravovanými zařízeními a monitorovacími systémy [19]. Díky této struktuře je možné snadno rozšiřovat monitorovací schopnosti o nové objekty, aniž by bylo nutné měnit základní architekturu systému.

Rozšíření RNM MIB je definované právě v RFC 1271. Toto RFC bylo nahrazeno novějšími [33], avšak formát výstupu z nástroje `nfb-eth` s nimi zatím kompatibilní není, zejména kvůli jinému datovému typu většiny statistik — RFC 2819 striktně zavádí `Counter32` (32bitový datový typ), položky výstupu typu `Counter` však mohou mít velikost 48-64 bitů [36].

[16] Management information base
[17] Simple Network Management Protocol
[18] Structure of Management Information

2.6.5 etherStats

Dříve zmíněný formát výpisu statistik z nástroje `nfb-eth` je nazván `etherStats`, což vychází z pojmu `etherStatsTable` definovaného v RFC 1271 [32]. Tato specifikace obsahuje řadu položek, které je možné v rámci RNM MIB sledovat. Hodnoty sledované v rámci `etherStats` mohou vypadat např. takto:

```
"etherStats": {
    "octets": 351232,
    "pkts": 5488,
    "broadcast": 5488,
    "multicast": 0,
    "crc_align_errors": 0,
    "undersize": 0,
    "oversize": 0,
    "fragments": 0,
    "jabbers": 0,
    "pkts64": 5488,
    "pkts65to127": 0,
    "pkts128to255": 0,
    "pkts256to511": 0,
    "pkts512to1023": 0,
    "pkts1024to1518": 0,
    "conf_undersize": 0,
    "conf_oversize": 0
}
```

Ve výpisu členů stejnojmenného objektu lze vidět množství informací, např. počet zpracovaných oktetů[19] s neceločíselnou velikostí v oktetech nebo s neplatným kontrolním součtem (FCS) [32] (položka `octets`), počet zpracovaných paketů (položka `pkts`), počty zpracovaných zástupců jednotlivých skupin (broadcast, multicast), počty rámců pod minimální a nad maximální velikost atp. Většinu těchto statistik lze testovat ve stávajícím testovacím prostředí, některé vyžadují nutnost jeho rozšíření (viz sekce 6.3.2).

2.7 Nástroj `ndk-nic-ctl`

Pro konfiguraci MVB paketového filtru designu NIC slouží nástroj `ndk-nic-ctl`, dostupný v rámci zdrojových kódů designu NIC. Nástroj je napsaný v Pythonu a kromě filtru umí konfigurovat i komponentu RSS [15, NDK-NIC-CTL tool]. Spuštěný bez parametrů vypisuje informace o firmware karty: počet aplikačních jader, nastavení registrů, čítače a aktuální konfiguraci filtru a RSS. Přepínačem `-d` je možné zvolit připojené zařízení, podobně jako u nástrojů popsaných výše. Přepínači `--filter_conf` a `--rss_conf` je volen soubor s konfigurací filtru, resp. RSS, který má být naparsován a načten do zařízení. Ukázkové spuštění může vypadat např. takto:

```
ndk-nic-ctl -i 0 --filter_conf filter.yaml
```

Kde přepínačem `-i` lze zvolit aplikační jádro. Soubor s konfigurací využívá syntaxi jazyka YAML.

[19] V kontextu počítačových sítí je často používán pojem oktet specifikující množinu přesně osmi bitů, v kontrastu s bajtem, kdy tento historicky znamenal i množiny 7 bitů

2.7.1 Jazyk YAML

YAML (z rekurzivního akronymu YAML Ain't Markup Language) je serializační jazyk pro data navržený tak, aby byl dobře lidsky čitelný, v kontrastu např. s jazykem JSON (2.6.3). Jeho hlavní předností je jednoduchost [34, kapitola 1]: na rozdíl od mnoha jiných jazyků podporuje pouze 3 základní datová primitiva: **mapování** (*slovníky*), **sekvence** (*pole, seznamy*) a **skalární hodnoty** (*textové řetězce, čísla, pravdivostní hodnoty*). Pomocí těchto datových struktur lze uložit prakticky libovolné hodnoty. YAML je podporován celou řadou programovacích jazyků, pro účel této práce je však důležitá podpora v jazyce Python[20].

Specifikace jazyka byla vytvořena v roce 2004 a prochází neustálým vývojem, s aktuální verzí 1.2.2 vydanou v roce 2021 [34, Status of this Document]. YAML je jazyk citlivý na odsazení. Každý záznam — uzel v grafu — je na svém vlastním řádku. Jednotlivé úrovně uzlů jsou odděleny odsazením, standardně dvěma mezerami (`'  '`)

Sekvence jsou uspořádané kolekce prvků. Každý prvek kolekce je uvozen znakem `-` a znakem mezery (`' '`). *Mapování* využívají dvojtečku a mezeru mezi každou dvojicí `klíč: hodnota`. *Komentáře* jsou označeny znakem mřížky (`#`).

YAML podporuje různé datové typy, jako jsou *pravdivostní hodnoty* (`true`, `false`), *čísla* (celá i desetinná) a *null* hodnoty (`null` nebo `' '`). Tyto datové typy jsou automaticky rozpoznány při parsování [34, kapitola 3.3.2]. Textové řetězce jsou psány bez uvozovek, pouze tam, kde se v řetězci vyskytuje znak dvojtečky, je nutné uvozovky použít, aby byl tento případ odlišen od začátku dalšího mapování.

2.7.2 Konfigurace filtru

Soubor s konfigurací pro MVB filtr může vypadat např. takto [17]:

```
filter:
  tcam_rules:
  - match:
      proto: 6
      sport: 39104
      sip: 1.1.1.1
    action:
      drop: 'true'
      redirect: 0
      mark: 5
```

Počátek konfigurace filtru značí uzel nejvyšší úrovně `filter`. Samotný výčet pravidel je poté v sekvenci `tcam_rules`. Každé pravidlo se skládá ze dvou částí: `match` — vlastnost filtrovaných paketů, při jejíž shodě má být použito — a `action` — akce, která se má s danými pakety provést. Mezi pravidly je uplatněn **logický OR** (|).

`match` podporuje následující položky:

- `ifc` – číslo síťového rozhraní,
- `vlan_tci` – číslo vnější[21] VLAN,
- `sip`, `dip` – zdrojová, resp. cílová IPv4/IPv6 adresa – zde je kromě délky prefixu podporována i bitová maska,

[20] Podpora YAML v Pythonu je dostupná prostřednictvím frameworku PyYAML [27]

[21] v tomto kontextu je pojmem *vnější* míněna VLAN nejvyšší úrovně, pokud je jich v jednom paketu více; např. při využití standardu 802.1Q-in-802.1Q

- `proto` – číslo protokolu transportní (L4) vrstvy,
- `sport`, `dport` – číslo zdrojového, resp. cílového portu některého z protokolů TCP, UDP či SCTP [17].

Cílová akce – `action` – filtr podporuje následující akce:

- `drop` – zahození paketu, který odpovídá kritériím
- `redirect` – přesměrování do zvoleného DMA kanálu (index)
- `mark` – označení paketu v poli `Filter bitmap` struktury NIC metadata header (4.2)

Přesměrování má prioritu před pravidly nakonfigurovanými v modulu RSS. Pokud není filtr nakonfigurován pomocí nástroje `ndk-nic-ctl`, použije se jeho výchozí chování:

```
drop: false
mark: 0
redirect: false
```

Tedy efektivně je filtr vypnutý. Vlastností i akcí může mít každé pravidlo samozřejmě více, v takovém případě se mezi jednotlivými prvky uplatní **logický AND** (&).

2.8 Testovací prostředí

Následující sekce vychází z dokumentace a kódu v repozitáři `hwtests` [14].

Sdružení CESNET v současnosti vyvíjí dvě prostředí pro testování svých síťových karet:

1. **ndktests** – starší řešení postavené na čistě vlastních zdrojových kódech, využívající starší verzi in-house frameworku `lbr_testsuite`. `ndktests` je dnes již zastaralé, nadále neudržované a používá se pouze v nástrojích pro automatickou integraci kódu[22] používaných ve sdružení CESNET. Postupně je nahrazováno druhým jmenovaným.
2. **pytest** – v současnosti vyvíjené a udržované sady testů postavené na frameworku `pytest`.

Obě tato prostředí jsou uchována, resp. vyvíjena rámci repozitáře `hwtests` v privátní instanci úložiště GitLab, které CESNET pro tyto účely provozuje [14].

Vývoj probíhá téměř výhradně na serverech sdružení, kde jsou potřebné nástroje a zejména také připojené síťové karty, pro které jsou testy vyvíjeny. Servery jsou dostupné v privátní síti projektu Liberouter prostřednictvím techologie *VPN*[23]. Před započetím vývoje je nejprve nutné připravit testovací prostředí pomocí skriptu `prepare_machine.sh` dostupného v kořenovém adresáři repozitáře:

```
$ ./prepare_machine.sh nazev_serveru
```

Skript využívá automatizačního frameworku Ansible[24] a vyžaduje tedy pro svůj běh přítomnost tohoto frameworku a několik dalších závislostí. Ty jsou dostupné na k tomu

[22] CI — continuous integration

[23] Virtual Private Network — technologie pro vzdálené připojení do vnitřní sítě organizace či firmy tak, aby bylo možné přistupovat k lokálním síťovým zdrojům.

[24] https://docs.ansible.com/ansible/latest/getting_started/introduction.html

určeném virtuálním serveru zvanému `ryzlink-vlassky`, kde je možné zajistit přístup k testovacím serverům použitím protokolu SSH[25] s využitím autentizačních klíčů.

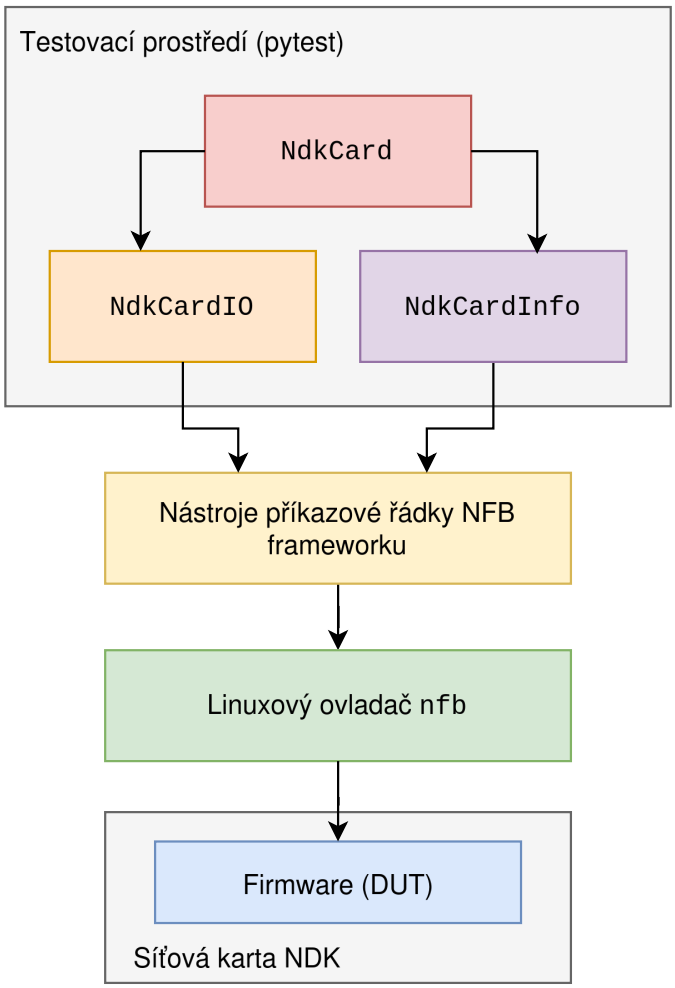
Po přípravě prostředí přes `ryzlink-vlassky` je opět využito SSH pro vývoj přímo na připraveném serveru. Po naklonování repozitáře `hwtests` a vstupu do složky `pytest` lze spustit testovací sady pomocí příkazu popsaného podrobněji v kapitole o frameworku pytest (později podrobně popsán v kapitole 3):

```
$ pytest
```

Spuštění bez parametrů vyvolá kolekci a spuštění testů automaticky omezených pouze na podmnožinu relevantní k výchozí připojené síťové kartě a nahranému designu. Ostatní testy jsou přeskočeny, což je zajištěno pomocí speciálních fixtures (rozebrány v sekci 3.3) a v nich použité konstrukce `pytest.skip()`. Dále je možné využít přepínače `--tested-pci-addr`, kterým je explicitně specifikováno zařízení, pro které mají být spuštěny testy:

```
$ pytest --tested-pci-addr 0000:3b:00.0
```

kde argument přepínače — PCIe adresu testovaného zařízení — je možné získat např. z výstupu příkazu `nfb-info` (popsán v podsekci 2.6.2) s použitím přepínače `-d`.



Obrázek 2.3: Architektura testovacího prostředí (vychází z [14])

[25] Secure Shell — dnes tzv. industry standard pro zajištění přístupu ke vzdáleným zařízením [29].

Testovací prostředí poskytuje množství objektů určených pro práci s kartou a jejím firmware. Na diagramu 2.3 lze vidět architekturu komunikace s kartou prostřednictvím tříd `NdkCard`, `NdkCardIO` a `NdkCardInfo`. Jejich implementace je v cestě `pytest/src` v souborech `ndkcardio/ndkcardio.py`, resp. `card/ndkcard.py` a `ndkcardinfo/ndkcardinfo.py`. `NdkCard` poskytuje vysokoúrovňové API pro práci s některými typy designů[26] (včetně designu NIC). Následně jsou prostřednictvím API tříd `NdkCardIO` a `NdkCardInfo` volány nástroje příkazové řádky frameworku NFB (zmíněn v sekci 2.6), které přes ovladač pro linuxové jádro přistupují přímo k registrům a dalším komponentám síťové karty. Tímto lze zajistit dostupnost všech funkcí potřebných pro testování přímo v prostředí pytest frameworku.

Funkčnost konkrétní síťové karty poskytuje z velké části právě její firmware. Proto je také hlavním předmětem testování a na obrázku je vyznačen jako komponenta uvnitř karty. Pojmem *DUT* se rozumí **Device under test**, což je označení produktu, který je v daném okamžiku podrobován testování. Méně známá jsou také označení *EUT* (Equipment under test) a *UUT* (Unit under test).

2.8.1 `NdkCardIO`

Tato třída slouží primárně jako wrapper[27] pro příkazy frameworku NFB. Kromě logování a uložené cesty k souboru zařízení síťové karty[28] nemá její instance prakticky žádné atributy a tedy neuchovává téměř žádný vnitřní stav. Za zmínku stojí způsob vnitřního volání příkazů, který využívá třídu `executable.Tool` z již dříve zmíněného frameworku `lbr_testsuite`. Vzhledem k výše uvedeným okolnostem lze tedy tuto třídu označit za poměrně nízkoúrovňovou abstrakci nad příkazy.

2.8.2 `NdkCard`

Třída `NdkCard` naproti tomu poskytuje vysokoúrovňové API pro některé složitější operace, mezi něž patří:

- konfigurace DMA kanálů pomocí metody `configure_dma_distribution()`,
- abstrakce nad nastavením příjmu příchozího provozu z karty[29] prostřednictvím metod `start_receiving()` a `stop()`,
- konfigurace RSS komponenty.

Třída dále dědí z abstraktní třídy[30] `Card`, která jí poskytuje implementaci API výše zmíněných tříd `NdkCardIO` a `NdkCardInfo`.

2.8.3 Knihovna Scapy

Pro generování paketů je v testovacím prostředí použita knihovna **Scapy** [28]. Scapy je komplexní program pro vytváření, analýzu, odchyt a podvrhávání paketů. Lze jej využít

[26] Nelze například využít pro design CyberThreats [31]

[27] Wrapper je kód, který abstrahuje API nějaké komponenty pro účely komponenty jiné — zde se jedná o volání nástrojů příkazové řádky s prefixy `nfb-` a `ndp-` z frameworku NFB.

[28] network card device path

[29] Mimo jiné za pomoci asynchronního spuštění příkazu `ndp-receive` — popsán v podsekci 2.6.2.

[30] ABC — Abstract Base Class

jako importovatelnou knihovnu k vytvoření široké škály síťových nástrojů včetně testovacích frameworků, i jako nástroj typu REPL[31] (např. v interaktivním příkazovém prostředí Python).

Knihovna podporuje široké spektrum protokolů napříč síťovými vrstvami. Implementační část mé práce ji primárně využívá pro vytváření testovacích dat. Vytvořit paket dle zadaných parametrů lze například takto:

```
from scapy.layers.inet import IP
ip_packet = IP()
```

Tímto způsobem je vytvořen IPv4 datagram s výchozími hodnotami. Ty jsou u IP vrstvy určeny routovací tabulkou, TCP má výchozí zdrojový port 20 a cílový 80, UDP zdrojový a cílový 53 a ICMP paket má typ `echo request`. Typy protokolů jsou u všech vrstev specifikovány vyšší vrstvou, pokud je přítomna.

Jednotlivým vrstvám lze samozřejmě změnit jejich výchozí hodnoty použitím správného parametru:

```
from scapy.layers.inet import IP
ip_packet = IP(src="1.1.1.1", dst="8.8.8.8")
```

Tímto lze i přepsat parametry předdefinované vyšší vrstvou a vytvořit tak nevalidní paket. Pro skládání více vrstev do jednoho datagramu lze využít tzv. *forward slash* operátoru:

```
from scapy.layers.l2 import Ether
from scapy.layers.inet import IP, TCP

packet = Ether() / IP() / TCP()
```

Tento operátor běžně znamená dělení, knihovní třída `Packet` však obsahuje privátní metodu `__div__`, která toto výchozí chování přepisuje [24]. Podobného konceptu využívá například knihovna `pathlib` pro skládání objektů typu `Path`[32].

Knihovna Scapy je v testovacím prostředí použita jako jedna z klíčových závislostí. Nejinak je tomu i u otevřeného frameworku pytest, který je popsán v následující kapitole.

[31] Read-Eval-Print Loop — iterativní vyhodnocování kódu, často se pro nástroje tohoto typu používá označení shell.

[32] Skládání objektů pomocí forward slash operátoru je vysvětleno v dokumentaci pro knihovnu `pathlib`: https://docs.python.org/3/library/pathlib.html#operators

Kapitola 3

Framework pytest

Následující sekce vychází z [11].

`pytest` je testovací framework pro jazyk Python, využívaný pro psaní jednotkových (unit), integračních a funkčních testů. Je široce používaný díky své jednoduchosti a čitelnosti. Testovací případy jsou napsány jako běžné funkce, doplněné případně o dekorátory[1]. Framework je také kompatibilní dalšími frameworky, např. `unittest`.

`pytest` je dostupný jako balík v balíčkovacím systému PyPi a lze jej nainstalovat pomocí

```
$ pip install pytest
```

Testovací prostředí CESNET v sobě zahrnuje prostředí pro Python, ve kterém je framework již připravený a nainstalovaný.

3.1 Použití

Balíček přidává do pracovního prostředí příkaz `pytest`. Ten je možné spustit kompletně bez parametrů, potom se použije výchozí chování — `pytest` má tzv. auto-discovery funkci implementující sadu pravidel nazvanou *Python test discovery* — při spuštění automaticky prohledá aktuální pracovní adresář a rekurzivně všechny podadresáře[2] a hledá v nich soubory s příponou `.py`, jejichž název začíná prefixem `test_` nebo končí sufixem `_test`.

Soubory je také možné specifikovat napřímo pomocí jednoho či více parametrů:

```
$ pytest test1.py test2.py
```

Poté jsou brány do úvahy pouze specifikované cesty (soubory, případně všechny soubory v adresáři, pokud se jedná o cestu k adresáři). V takto automaticky nalezených či explicitně specifikovaných souborech jsou následně vyhledávány funkce začínající prefixem `test` a všechny metody tříd začínající prefixem `Test`. Tyto jsou poté považovány za testovací funkce, jejichž tělo je kód daného testu.

Při kolekci mohou být testy také **omezeny** na konkrétní podmnožinu či úplně **vyřazeny**. Prvního chování lze docílit pomocí přepínače `-k`, následovaného case-insensitive výrazem vyhodnotitelným v syntaxi jazyka Python [20]:

```
$ pytest -k "apple and (not banana or cherry)"
```

[1] Dekorátor je v jazyce Python řádek zdrojového kódu vyskytující se u deklarace či definice funkce, metody či třídy, který upravuje její chování nebo účel.

[2] Pokud toto chování není upraveno konfiguračními soubory či proměnnými prostředí, viz dále.

Výše zadaný parametr způsobí vybrání pouze takových testovacích případů, které ve svém plném názvu[3] obsahují řetezec `apple` a zároveň: 1. neobsahují řetězec `banana` nebo 2. obsahují řetězec `cherry`.

Vyřadit, nebo také přeskočit test je možné kdekoliv v jeho těle nebo požadovaných závislostech pomocí volání `pytest.skip()` s případnou zprávou jako volitelný argument.

Výchozím chováním frameworku po dokončení fáze kolekce je automatizované sériové spuštění nalezených testovacích funkcí. Specifikací přepínače `--collect-only` (též `--co`) je možné toto chování změnit a nechat tak pouze vypsat jejich seznam spolu s jejich komentáři. Při spojení s přepínačem `-q` je vypsán pouze seznam testů bez komentářů [20]:

```
$ pytest --co -q
```

Posledním důležitým přepínačem je `--pdb`, který vyvolá *Python debugger*[4] při prvním selhaném testu nebo přerušení běhu testů pomocí sekvence Ctrl+C (jinými slovy, při vyvolání jakékoli výjimky v průběhu spuštění).

Konfigurace je dále možná pomocí proměnných prostředí a definic v souborech `pytest.ini` a `pyproject.toml`.

3.2 Životní cyklus testu

V nejjednodušším slova smyslu je test určen k ověření výsledku chování určité komponenty či software a jestli toto chování odpovídá očekáváním [21, Anatomy of a test]. Chování jako takové nelze empiricky změřit, a proto vývoj testů může být mnohdy náročná disciplína.

Životní cyklus testu můžeme rozdělit do čtyř fází:

1. **Arrange** – příprava prostředí pro samotný test. Toto obnáší např. inicializaci potřebných objektů, přípravu testovacích parametrů apod. Může to znamenat např. inicializaci fixtures (3.3). Ideálně by mělo být vykonáno vše, co neobnáší následující bod (Act), aby byl test co nejvíce izolovaný a průkazný.

2. **Act** – vykonání samotné akce, která nějakým způsobem mění stav systému, který bude následně vyhodnocen. Typicky se jedná o volání metody/funkce, např. konfigurace filtru a odeslání paketů.

3. **Assert** – v tomto kroku se vyhodnocuje změněný stav. Touto změnou může být například zvýšení počtu paketů přijatých síťovou kartou nebo naopak počet paketů zahozených.

4. **Cleanup** – Systém by po své změně a jejím vyhodnocení ideálně měl být opět navrácen do původního stavu pro následující testy. To může obnášet například částečný restart[5] karty, aby byly vynulovány čítače.

Body 1 a 3 staví převážně na schopnostech testovacího frameworku, a proto budou dále rozebrány podrobněji.

[3] ten se skládá z názvu souboru, názvu testovací funkce a názvů případných parametrů
[4] pdb — název ladicího nástroje integrovaného přímo v interpretu jazyka Python
[5] warm boot; bez odpojení napájení

3.3 Fáze arrange

Framework pro fázi **arrange** poskytuje vhodné prostředky — tzv. *testovací fixtures*. Následující sekce proto vychází z [21, About fixtures].

V testovacích nástrojích často vzniká potřeba vytvořit a opakovaně používat konzistentní a izolované prostředí pro spouštění jednotlivých testů, což je důvod, proč se používají testovací fixtures. Ty mají často podobu předem (před během testu samotného) vytvořených instancí objektů držících datové struktury či konkrétní stav okolních komponent. Ve frameworku pytest se fixtures vytvářejí pomocí funkcí se speciálními dekorátory.

K vytvořené fixture lze poté v konkrétním testu přistoupit pomocí argumentu funkce se stejným názvem jako fixture. Pokud je funkce v aktuálním kontextu viditelná, `pytest` ji sám vykoná a její návratovou hodnotu použije jako parametr. Fixtures podporují princip DRY (don't repeat yourself). Tento přístup také automaticky využívá cachovací mechanismy frameworku [21, About fixtures — Sharing test data]. Následuje ukázka použití fixtures v praxi:

```
import pytest
import logging

class NetworkCard:
    def __init__(self, addr):
        self.addr = addr

    def __str__(self):
        return f"Network card with address {addr}"

@pytest.fixture
def net_card():
    return NetworkCard()

def test_net_card_capabilities(net_card):
    logging.info(net_card)
```

Po importu potřebných modulů je zde vidět definice třídy `NetworkCard` reprezentující imaginární síťovou kartu. Ta obsahuje dvě metody: konstruktor (v Pythonu nazvaný klíčovým slovem `__init__`) s argumentem adresy, která má být kartě nastavena, a speciální metodu `__str__`, jež je zavolána při použití objektu v řetězci. Následuje deklarce fixture — funkce s dekorátorem `pytest.fixture` —, která pouze instanciuje a vrátí instanci dříve zmíněného `NetworkCard` objektu.

Nakonec je definováno samotné tělo testovací funkce, jejíž API vyžaduje název funkce `net_card` jako svůj argument. Tím je uvnitř funkce `test_net_card_capabilities` zpřístupněn objekt `NetworkCard` a ten je pro jednoduchost pouze vypsán pomocí modulu `logging`.

3.4 Fáze assert

`pytest` téměř výhradně pro tuto fázi (určení výsledku testu) využívá standardního klíčového slova `assert` jazyka Python. Vzhledem k přehledné syntaxi může být kód využívající toto klíčové slovo z velké části sebe-dokumentující. Výraz `assert` lze zapsat dvěma způsoby [23]:

1. `assert výraz`, který je zjednodušeně ekvivalentem:

   ```
   if not výraz:
       raise AssertionError
   ```

2. `assert výraz1, výraz2`, který navíc předává konstruktoru výjimky `výraz2` jako argument:

   ```
   if not výraz1:
       raise AssertionError(výraz2)
   ```

V obou případech platí, že pokud se `výraz1` vyhodnotí na hodnotu branou v booleovském kontextu jako `False` (nepravda), běh daného testovacího případu je ukončen vyvoláním výjimky `AssertionError`, kterou není třeba odchytávat a při jejíž propagaci až na návrat z testovací funkce je vypsán přehledný popis nežádoucího výsledku fáze Act (vysvětlena v sekci 3.2):

```
def test_func(zero=0, one=1):
> assert zero == one
E assert 0 == 1

test.py:2: AssertionError
```

V příkladu výše jsou proměnné funkce `test_func` naplněny hodnotami 0 a 1 a poté jsou testovány na shodu, což vyústí ve výše zmíněnou chybu. Těchto *asercí* může být v rámci jednoho běhu testu libovolný počet a lze tak například kontrolovat některé vedlejší efekty testovaného chování.

3.5 Parametrizace

Další schopností frameworku je možnost parametrizace testů. Ta umožní psát vesměs generický kód testu, který je následně za běhu doplněn statickými či podle určitých pravidel sestavenými parametry. Parametrizace se opět provádí pomocí dekorátoru. Ten Python podporuje od verze 2.4 pro funkce a metody a od verze 2.6 poté i pro třídy [22].

Dekorátor pro parametrizaci je uvozen jeho názvem `pytest.mark.parametrize`. Přijímá množství argumentů, z nichž podstatné a povinné jsou dva: název proměnné, do které se v každé iteraci uloží hodnota a následně výčet těchto hodnot jako `Iterable`[6].

Následující kód ilustruje využití parametrizace na jednoduchém příkladu:

```
import pytest

parameters = [("Apple", 5), ("Banana", 6)]

@pytest.mark.parametrize("name_with_length", parameters)
def test_if_length_matches(name_with_length):
    name, length = name_with_length
    assert len(name) == length
```

[6] `Iterable` je objekt schopný vracet své členy jeden po druhém. Příklady iterovatelných objektů zahrnují všechny sekvenční typy (například `list`, `str` a `tuple`) a některé nesekvenční typy, jako je `dict`, objekty souborů (`file objects`) a objekty tříd, které definují metodu `__iter__` nebo metodu `__getitem__`, jež implementují sekvenční sémantiku [26].

V ukázce výše můžeme vidět definici seznamu parametrů, který bude následně vstupovat do dekorátoru `parametrize`. Seznam má dva členy, v tomto případě n-tice (tuple), kde prvním prvkem je řetězec reprezentující název ovoce a druhý prvek je poté délka tohoto řetězce. Dekorátor seznam rozdělí na jednotlivé členy a následně iterativně volá dekorovanou funkci a předává jí postupně každý z nich jako hodnotu argumentu s názvem `name_with_length`. Uvnitř funkce je n-tice rozdělena na prvky *name* a *length*. Ty jsou prostřednictvím klíčového slova `assert` a využití funkce `len()` následně zkontrolovány na shodu.

Parametrizaci lze u jedné testovací funkce použít i vícenásobně (řetězením[7] dekorátorů použitých nad funkcí), a vytvořit tak maticové testy.

3.5.1 Maticové testy

Tento pojem není zatím příliš znám v odborné literatuře, nicméně je hojně používán v mnoha komerčních nástrojích pro automatickou integraci kódu a testovacích frameworcích při sestavování či testování softwaru. V zásadě se jedná o kartézský součin množiny vstupních parametrů se sebou samou, čímž vzniknou jejich všechny možné kombinace. Těmito parametry mohou být (při sestavování softwaru) například instrukční sada procesoru nebo cílový operační systém [9]. V oblasti testování softwaru se pak může jednat o kombinace vstupních parametrů testu, což ilustruje následující příklad: Uvažujme množiny vstupních parametrů Fruit $= \{\text{Apple}, \text{Banana}, \text{Cherry}\}$ a Count $= \{1, 2, 3\}$, potom jejich kartézský součin ilustruje tabulka 3.1.

Fruit/Count	**1**	**2**	**3**
Apple	(Apple, 1)	(Apple, 2)	(Apple, 3)
Banana	(Banana, 1)	(Banana, 2)	(Banana, 3)
Cherry	(Cherry, 1)	(Cherry, 2)	(Cherry, 3)

Tabulka 3.1: Matice vstupních parametrů testu

Každá z těchto kombinací poté představuje samostatnou n-tici vstupních parametrů pro testovací funkci.

[7] V tomto kontextu je řetězením myšleno prosté uvedení více dekorátorů na řádcích pod sebou.

Kapitola 4

Návrh a implementace testovacích sad

Naprostá většina výše zmíněných principů a nastudované dokumentace byla následně využita v implementační části této práce. Jejím úkolem bylo navrhnout a poté implementovat dvě testovací sady pro zvolenou funkcionalitu síťových karet CESNET.

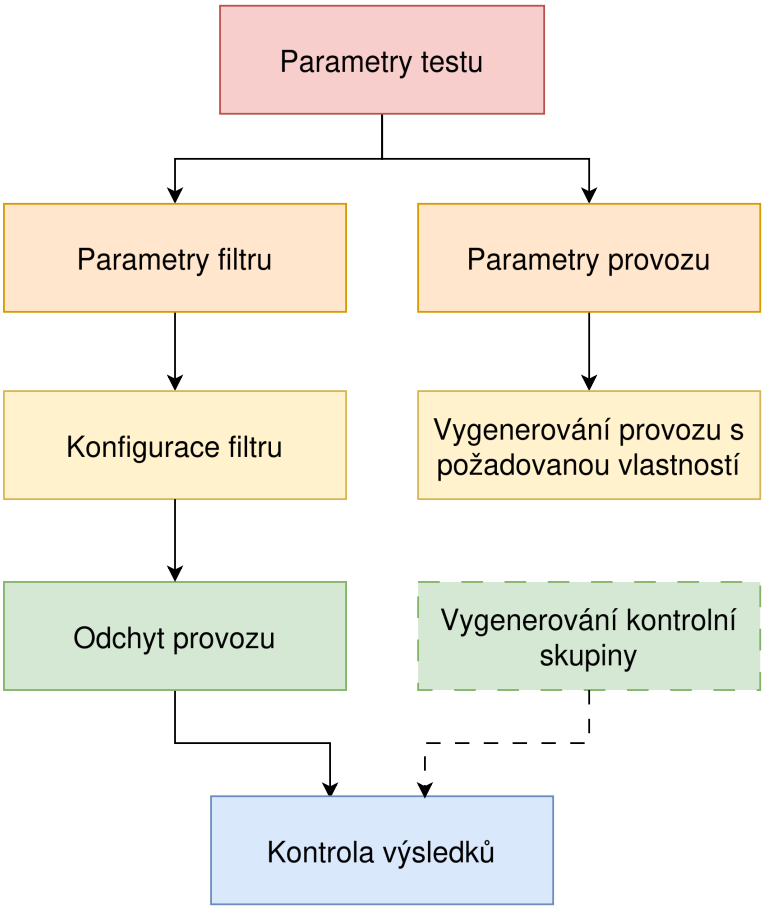
4.1 Návrh

Po dohodě s odborným konzultantem byly identifikovány dvě oblasti, na něž by měly být testy zaměřeny:

- v prvním případě se jedná o **MVB filtr designu NIC**,
- v druhém případě o **komponentu MAC** přítomnou v síťovém modulu většiny karet.

V obou případech se jedná o nějakou formu filtru, proto dává smysl testovat především filtrování provozu. Testovací sady se proto převážně zaměřují na vysílání skupin paketů, které pomocí v kartě nastaveného loopbacku dorazí zpět.

U obou testovacích sad proběhl návrh jak testovaných případů, tak samotného životního cyklu testů. Konkrétní testované případy jsou rozebrány v implementační části, průběh životního cyklu testů lze vidět na diagramu 4.1.



Obrázek 4.1: Návrh životního cyklu testu

Všechny testy v zásadě dodržují životní cyklus testu z kapitoly 3.2. Nejdříve proběhne inicializace parametrů testů a potřebných objektů pomocí fixtures. Některé z těchto fixtures nejsou přítomné v testovacím prostředí a proto je nutné je doimplementovat. Testovaný případ se vždy skládá z konfiguračních parametrů a parametrů pro generovaný provoz. Následuje samotné tělo testu, kde proběhne:

1. **inicializace** karty, zejména PMA loopbacku pro zpětný příjem odeslaných paketů,
2. **konfigurace** karty, zejména nastavení filtru,
3. **nastavení odchytu** provozu[1],
4. **odeslání** skupiny testovacích paketů s danou vlastností,
5. (volitelně) **odeslání kontrolní skupiny**[2]
6. **vyčkání** na doručení všech paketů na RX rozhraní a
7. **kontrola** čítačů přijatých dat.

[1] Pouze u testů NIC filtru.

[2] V případě, že se jedná o testovací případ pro zahození paketů, je třeba vygenerovat i kontrolní skupinu, kterou pravidlo nezahodí, aby bylo možno zjistit, že předchozí skupina byla korektně zpracována.

4.1.1 Randomizace

Důležitou vlastností testů je **randomizace**. Tam, kde to bylo vzhledem k testované funkcionalitě možné, bylo učiněno rozhodnutí testovací parametry náhodně generovat. U testování 3. vrstvy TCP/IP modelu se jedná o náhodné generování masek podsítě, velikostí prefixů a samotných IP adres. U 4. vrstvy jsou těmito náhodnými parametry čísla portů transportních protokolů. V případě testování MAC je pro změnu vhodné generovat náhodné MAC adresy. V některých případech tyto hodnoty závisí na povaze testu, například u testů multicast provozu je zapotřebí generovat podle konkrétní šablony či prefixu, aby daná adresa měla stále povahu multicastové adresy.

4.2 Implementace

Jak již bylo zmíněno, praktická část této práce se sestává z implementace dvou testovacích sad. Obě testovací sady jsou — s výjimkou jednoho doplňku — implementovány v jazyce Python s využitím frameworku `pytest` a dalších knihoven a nástrojů dostupných v testovacím prostředí. Jedná se primárně o testy zaměřené na funkcionalitu, nikoliv o zátěžové (stress) či jiné typy testů. Většina testovacích případů se soustředí na ověření jedné konkrétní sady parametrů, které nastavují určitou funkcionalitu, a následné zhodnocení stavu síťové karty po jejich aplikaci. Ve výjimečných případech, kdy to povaha testované funkcionality vyžaduje, obsahují testy více asercí. Tyto případy jsou v příslušných sekcích explicitně zmíněny.

Testovací sady nejsou samostatně funkční a pro jejich spuštění je nezbytné zařadit jejich soubory na správná místa, aby mohly být spuštěny spolu se zbytkem kódu testovacího prostředí. Všechny soubory s čistě vlastním kódem autora jsou formátovány na maximální délku řádku 80 znaků pro dobrou čitelnost v menších terminálových oknech.

4.3 Testovací sada pro NIC filtr

První sada testů je zaměřena na filtr v designu NIC. Tato sada je implementována v souboru `pytest/nic/functional/test_nic_filtering.py`. Pomocné funkce a deklarace lze poté najít v souboru `pytest/nic/functional/helper.py`. Pro testování funkcionality `mark` byla implementována pomocná aplikace v jazyce C.

4.3.1 `helper.py`

V tomto souboru existují pomocné funkce a deklarace pro první testovací sadu. První důležitou zajímavostí je definice vlastních datových typů pomocí enumerace:

```
class FilterActionName(StrEnum):
    Drop = auto()
    Redirect = auto()
    Mark = auto()
```

Enumeraci (česky výčet), podporovanou od verze 3.4, lze v Pythonu vytvořit pomocí třídy dědící od třídy `Enum`. Enumerace je množina prvků se symbolickými jmény přiřazenými ke konkrétním hodnotám, nad nimiž lze jednoduše iterovat [25]. V tomto případě je použita odvozená třída `StrEnum`, jejíž velkou výhodou je automatická konverze jména na

textový řetězec při použití instance objektu jako součást textového řetězce. Členové zde mají přiřazenu návratovou hodnotu funkce `auto()`, která slouží jako placeholder v datových typech dědících od `Enum` a jejích potomků, aby se ze syntaktického hlediska jednalo o definici. Tento princip se v implementaci vyskytuje na několika dalších místech a podporuje udržitelnější kód, protože definice jsou často na jednom místě a zároveň vhodně pojmenované, bez použití magických hodnot.

V tomto případě je enumerace využita pro reprezentaci 3 různých akcí, které podporuje komponenta filtru v designu NIC: `drop`, `redirect` a `mark` (podrobněji popsány v sekci 2.4). Tento kód poté následují definice protokolů 3. a 4. vrstvy modelu TCP/IP, které jsou opět výčtem podporovaných protokolů v pravidlech filtru.

Další zajímavostí je definice datových typů pro anotace. Zde se vyskytuje definice pomocí třídy `Callable`:

```
GenFunc = Callable[[NdkCardIO, Path, list[int]], None]
```

Tento řádek kódu popisuje datový typ funkce, kterou lze zavolat. Dle gramatiky anotací pomocí Callable[3]

je vždy první hodnota seznam typů argumentů funkce a druhá její návratový typ. Ten je `None`, funkce tohoto typu mají pouze vedlejší efekt a nic nevrací. Zde se jedná o obecný popis funkce pro generování paketů.

Tyto definice jsou pak následně použity pro vytvoření vlastních datových typů `TestParams` a `TestCase`. Zde je využit datový typ `tuple` (n-tice) pro kompozici více hodnot dohromady:

```
TestParams = tuple[ConfFunc, GenFunc, bool]
```

TestParams slouží jako množina parametrů pro testovací funkci, kdy `ConfFunc` je funkce s předvyplněnými parametry pro nastavení NIC filtru, `GenFunc` je předvyplněná funkce pro generování odpovídajících paketů a hodnota typu `bool` označuje, zda pro danou množinu paketů platí podmínka v pravidlu, a tedy má být na ně uplatněna daná akce, či nikoliv (True = match, False = miss).

Soubor dále obsahuje množství pomocných funkcí pro práci s IP adresami a testovacími parametry. Funkce `rand_ip_from_mask`, jak už její název napovídá, je určena pro generování náhodné IPv4, resp. IPv6 adresy dle zadaných pravidel (konkrétně síťové masky). Pro IPv4 zde lze využít funkci knihovny Scapy, pro protokol IPv6 musel být pro generování dle masky využit jiný přístup s pomocí knihovny `ipaddress`.

Pro manipulaci s IP adresami jsou zde dále dvě funkce pro konverzi IP adresy z číselného (`int`) do dotted-decimal formátu a zpět a funkce pro nulování části pro identifikaci zařízení (hosta), čehož je následně užíváno při generování IP adres náležících do dané podsítě.

Jako poslední 3 pomocné funkce se zde nachází `get_test_params`, `get_test_names` a `sort_dict_by_key`, které slouží jako drobná abstrakce nad některými běžně prováděnými operacemi, což na několika místech zvyšuje čitelnost kódu.

4.3.2 `test_nic_filtering.py`

Tento soubor je jádrem pro první sadu testů. Obsahuje několik funkcí pro abstrahování práce s kartou a následně 3 testovací funkce, z nichž 2 jsou parametrizované. Na začátku se nachází globální instanciace logovacího nástroje:

[3] Popis gramatiky je dostupný v dokumentaci modulu `typing` jazyka Python: https://docs.python.org/3/library/typing.html#annotating-callable-objects

```
logger = logging.getLogger(__name__)
```

Ten následně agreguje logovací výpisy pod jeden název (modulu) a je také předáván některým dalším funkcím, které ho dále používají. Poté následují účelné deklarace konstant. `CTL_PKT_CNT` slouží pro konfiguraci počtu odesílaných paketů, čímž lze tento nastavit globálně pro všechny testy v souboru. `MAX_MARK_BINS` je poté konstanta vycházející z velikosti pole Filter bitmap ve struktuře 4.2, kterou nelze jinak spolehlivě zjistit a tedy musí být přímo v kódu.

Většině pomocných funkcí je vždy předáváno několik společných parametrů, které funkce volaná jako samotný test dostane jako fixture (rozebrány v sekci 3.3) a ty jsou dále propagovány až k voláním funkcí testovacího prostředí.

Funkce `conf_filter` se stará o sestavení konfiguračního souboru pro NIC filtr (popsaný v sekci 2.7.2). Zde je dobré zmínit, že doplňková aplikace pro testování funkcionality `mark` je napsána způsobem, aby nemusela otevírat a alokovat všech 16, resp. 32 na kartách běžně dostupných front. Proto je kontrolována pouze fronta 0 a vzniká tedy potřeba zajistit, aby při značkování paketů byly tyto vždy zároveň přesměrovány na DMA kanál 0. Toho je docíleno tím, že se k action části nastavovaného pravidla `mark` přidá i akce `redirect` s hodnotou `0`:

```
if action_name == FilterActionName.Mark:
    action_rule["action"] |= {"redirect": 0}
```

Podobně je tomu ještě o pár řádků níže, kdy je přidáno ještě výchozí pravidlo pro případ paketů, na které filtr nemá mít vliv:

```
if action_name == FilterActionName.Mark:
    filter["filter"] |= {"default": {"action": {"redirect": 0}}}
```

Tento drobný detail nepatrně porušuje izolovanost testů, jelikož výsledek testu jedné funkcionality závisí na funkčnosti funkcionality jiné. Ta je nicméně testována taktéž a to v testech spouštěných dříve, není to tedy velký problém.

Následně je otevřen soubor pro uložení nastavení, které je do něj zapsáno a soubor je nahrán do karty:

```
with open(filter_conf_file, "wt") as filter_conf:
    yaml.dump(filter, filter_conf)


cardio.nic_filter_conf(app_core_index, filter_conf_file)
```

Následuje několik funkcí pro generování paketů s různými požadavky: `send_pkts` slouží pouze jako wrapper pro multiplikaci skupiny paketů na zadaný počet (ve výchozím stavu 32), aby toto nastavení bylo na jediném místě. `gen_control_group` odesílá kontrolní skupinu paketů — obsahujících pouze ethernetovou hlavičku —, s jiným počtem paketů, aby bylo snadné odlišit jednu skupinu od druhé. Další tři funkce s prefixem `gen_` slouží jako abstrakce pro generování paketů s konkrétní nastavenou vlastností.

Zajímavou funkcí, která stojí za rozebrání, je poté `check_received_packets`. Ta obsluhuje u drtivé většiny testů první sady 3. fázi (assert) životního cyklu testu (sekce 3.2). Funkce nejprve vyčte statistiky DMA kanálů pomocí k tomu určené metody z `NdkCardIO` a následně ověří, zda odpovídají očekáváním. Funkce je napsána natolik obecně, aby se dala použít jak pro porovnání souhrnných statistik (klíčové slovo "all"), tak i statistik konkrétních indexů. K tomu je použito obyčejné větvení.

Dále již následuje samotná definice testovaných případů. Ty jsou ukládány do seznamu (datový typ `list`) se speciálním datovým typem `TestCase`, definovaným v pomocném sou-

boru. Seznam je neprve definován jako prázdný (za dvojtečkou následuje anotace a přiřazení hodnoty "prázdný seznam"):

```
test_cases: list[helper.TestCase] = []
```

Následně je postupně plněn různými typy hodnot podle testované problematiky. Toho je docíleno pomocí několika iterací, kdy v každé iteraci následuje jedno nebo více volání `test_cases.append()`. To přidá callbacky na funkce — částečně naplněné argumenty — pro generování paketů a pro konfiguraci filtru, které jsou později v těle testu doplněny o fixtures potřebné pro běh testu.

Nejprve je testována 3. síťová vrstva (L3 v modelu TCP/IP) s protokoly IPv4 a IPv6. Pro oba protokoly je vygenerován náhodný prefix a náhodná adresa sítě (`rand_net`), jejíž generování je docíleno vygenerováním náhodné adresy a následně nulováním její hostovské části. Poté jsou pomocnou funkcí `get_bound_ip_values` přes její klíčové slovo `yield` postupně generovány konkrétní IP adresy pro různé scénáře: dolní a horní hranice intervalu síťových adres, náhodná adresa v intervalu a náhodná adresa mimo interval. `yield` vytvoří z funkce tzv. generátor [4], který si ukládá vnitřní stav a při opakovaném volání pokračuje v provádění svého kódu. Přitom přidává testovací případy jak pro zdrojovou, tak i cílovou adresu, což vychází z funkcionality podporované filtrem (sekce 2.7).

Dále je testovací sada rozšířena o testy transportní vrstvy (L4) s podporovanými protokoly TCP, UDP a SCTP[5]. U těch jsou vždy zvoleny dva náhodné porty a jejich různé kombinace: zdrojový a cílový port, se specifikací a bez specifikace konkrétního protokolu a zda je číslo portu shodné, či nikoliv. Z těchto parametrů jsou vyrobeny všechny možné kombinace a jsou přidány k testovaným případům.

Samotné testovací funkce jsou pak v této sadě tři: `test_nic_filter_drop_redirect`, `test_nic_filter_mark` a `test_nic_mark_invalid`. Těla prvních dvou testů jsou shodná s návrhem, kdy nejprve jsou parsovány parametry testu, poté proběhne inicializace karty včetně případných nastavení filtru, následně je do inicializované karty odeslán testovací provoz a nakonec jsou zkontrolovány čítače DMA kanálů. Poslední testovací funkce je speciálně zaměřena na objevenou chybu v nástroji `ndk-nic-ctl`, která je detailněji popsána v sekci 6.1. Ta je zaměřena přímo na chybnou konfiguraci filtru a proto nevyžaduje odeslání testovacího provozu, je totiž očekáváno, že fáze konfigurace nebude úspěšná.

4.3.3 Pomocná aplikace `nic-meta-mark`

Aplikace nazvaná `nic-meta-mark` slouží jako doplněk pro testování funkcionality `mark`. To totiž, vzhledem ke své povaze, vyžaduje přímý přístup k proprietárním metadatům sledovaných paketů, protože hlavičky běžných protokolů tyto informace neobsahují. Jak bylo zmíněno v sekci 2.7.2, cílová akce `mark` označí daný paket nastavením jednoho z bitů v poli `Filter bitmap` (bity 96-111). Toto pole je tedy při kontrole nutné extrahovat právě z metadat.

Kód aplikace a přidružené soubory lze nalézt v adresáři `/tools/nic-meta-mark`. Stěžejní část funkcionality je poté implementována v modulu `nic-meta-mark.c`. V adresáři se dále nachází 3 hlavičkové soubory s:

1. definicí maker pro ladicí výpisy v souboru `debug.h`,
2. určením návratových hodnot v souboru `error.h` a

[4] korutinu, též koprogram
[5] Stream Control Transmission Protocol

3. deklarací struktury `ndp_pkt_hdr` v souboru `ndp-packet.h`.

Posledním důležitým dílkem je soubor `Makefile` pro snadné přeložení do strojového kódu. Ten definuje cíle pro program Make: `compile` pro zahájení kompilace, `clean` pro vyčištění adresáře od vygenerovaných artefaktů (binárních souborů) a `run` pro ukázkové spuštění. Celý proces od kompilace po spuštění tedy může vypadat takto:

```
cd tools/nic-meta-mark
make compile
./nic-meta-mark /dev/nfb0
```

Program při svém spuštění (ve funkci `main` v souboru `nic-meta-mark.c`) nejprve otevře soubor se zařízením — specifikovaný jako první parametr — prostřednictvím volání funkce `nfb_open`. Po získání struktury popisující dané zařízení je otevřena jeho nultá fronta[6] pro příjem paketů (RX).

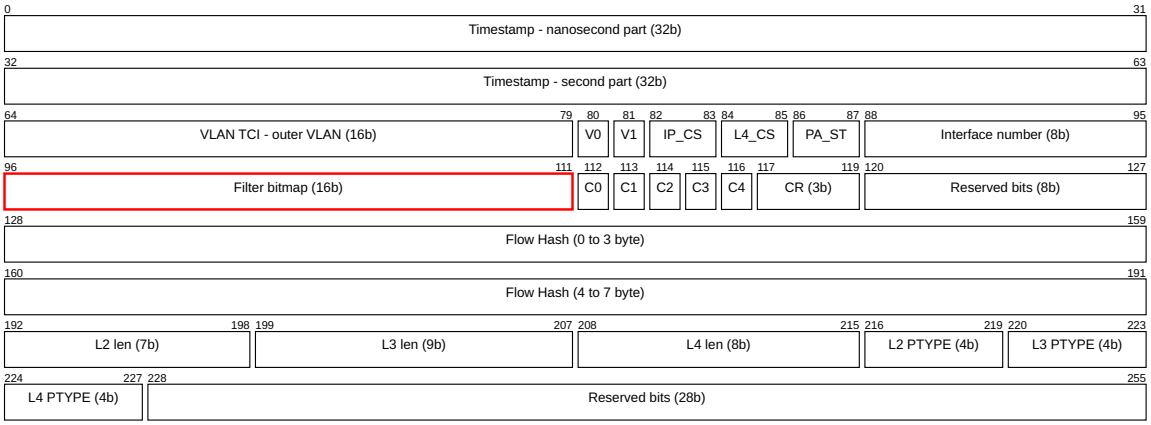
Dále následuje běžný běh programu, kdy jsou iterativně z fronty vyčítány shluky paketů (bursts) do staticky alokovaného pole. U každého načteného paketu je poté na standardní vstup vypsáno jeho pořadové číslo a bity, které jsou v bitmapě v logické jedničce:

```
$ ./tools/nic-meta-mark/nic-meta-mark /dev/nfb0
[0] 4
[1] 4
[2] 4
[3] 4
...
```

Takto jednoduchá práce s pakety v jazyce C je možná díky přetypování bufferu pro NIC packet metadata na strukturu `ndp_pkt_hdr`. Tu lze v podobě paketového diagramu vidět na obrázku 4.2 s vyznačením umístění položky `Filter bitmap`:

```
struct ndp_pkt_hdr * header = (struct ndp_pkt_hdr *) pkts[i].header;
```

Tento princip zkracuje kód a činí jej přehlednějším. Struktura musí být deklarována s klíčovým slovem `__attribute((packed));`, aby ji překladač nezarovnal a nezměnil tak význam mapování jednotlivých částí paketu na prvky struktury.



Obrázek 4.2: Hlavička s metadaty NIC

[6] v tomto případě jsou indexy front shodné s DMA kanály, tedy nultá fronta = DMA kanál 0 atd.

Všechny operace jsou kontrolovány na potenciální chybu (návratový kód funkce) a v takovém případě je program korektně ukončen s chybovou hláškou a odpovídajícím chybovým kódem definovaným v `error.h`. Stejně tak se stane i při běžném ukončení programu. Tomu napomáhá i definice funkce pro odchyt signálů `SIGTERM` a `SIGINT`, při jejichž vyvolání jsou uvolněny držené zdroje a aplikace končí s nulovým návratovým kódem. Toho poté využívá wrapper této aplikace v `pytest` části.

4.3.4 Wrapper pro `nic-meta-mark`

Pro výše zmíněnou aplikaci `nic-meta-mark` byla také vytvořena mezivrstva — přítomná v cestě `src/apps/ndk_nic/meta_mark/` —, která zajišťuje zejména dvě věci:

1. abstrakci nad spouštěním aplikace a získáváním jejích výstupů,
2. automatickou kompilaci C aplikace před během testů, které ji vyžadují,

Pro propojení API pomocné aplikace a rozhraní Python bylo nutné naimplementovat třídu `MarkParser` v souboru `mark_parser.py`. Ta je složena z konstruktoru, metod `run` a `stop` a tzv. *finalizeru*. Názvy prvních dvou metod jsou nejspíše sebe-dokumentující, *finalizer* slouží ke garanci ukončení aplikace v případě předčasného nebo nesprávného ukončení testů.

Druhý bod je vyřešen s využitím fixtures (soubor `fixtures.py`). První z nich, privátní funkce `_nic_meta_mark_path`, slouží pouze jako abstrakce pro cestu k souborům, podstatná je druhá funkce — `nic_meta_mark`. Díky modulárnosti frameworku pytest je možné fixtures řetězit, čehož je zde využito vytvořením závislosti na první funkci pomocí parametru. To poskytuje jisté výhody, zejména zpřehlednění kódu. V samotném těle funkce proběhne volání programu `make`, které díky zřetězení cílů `clean` a `compile` zajistí vždy rekompilaci pro nový běh testů, aby se předešlo využití případné starší verze aplikace. Následně je instanciován objekt MarkParser pro práci s aplikací a ten je vrácen jako parametr funkce, která si fixture vyžádala.

4.4 Testovací sada pro komponentu MAC

Druhá testovací sada je implementována zejména v souboru `common/eth/test_mac.py`. Některé nízkoúrovňové části komunikující přímo s nástroji frameworku NFB pak byly přidány do třídy `NdkCardIO` (podsekce 2.8.1). Sada obsahuje některé funkce pro generování provozu a kontrolu statistik podobné těm z první sady, ty zde proto nebudou dále rozebírány.

U komponenty MAC bylo nejprve testováno filtrování MAC adres. U filtrace MAC adres se jedná hlavně o testování všech čtyř režimů komponenty (sekce 2.5): *standardní režim*, *promiskuitní režim*, *multicastový režim* a režim *broadcast*. Pro všechny tyto režimy byly implementovány sledované skupiny MAC adres (seznam `mac_pref`) i testovací funkce:

1. `test_promisc_mode` – ověřuje, zda jsou při nastavení promiskuitního režimu propuštěny všechny sledované skupiny,
2. `test_normal_mode` – nastaví standardní režim a testuje, zda byl zahozen provoz s neodpovídajícími MAC adresami,
3. `test_mcast_mode` – ověřuje, zda nastavením multicastového režimu projdou pouze nastavené MAC adresy a multicastový provoz,

4. `test_bcast_mode` – implementuje to stejné pro broadcast.

Druhá část testování je zaměřena na velikosti paketů[7] — min frame length (minimální délka rámce) a max frame length, též *MTU* (Maximum Transmission Unit). Pro tuto část je definován konstantní seznam `MTU_TEST_PARAMS`, naplněný různými částmi intervalu

$$(MIN\_FRAME\_LEN,\ MAX\_FRAME\_LEN]$$

testující vždy hodnotu samotnou a také její blízké sousedy, tedy hodnotu o jedna menší a hodnotu o jedna větší. Hodnoty byly zvoleny pro odhalení *off-by-one* chyb, běžných v programech.

Třetí část testovací sady pro MAC se zabývá ověřením chování čítačů při vypnutém režimu vysílání, příjmu či obojího. Pomocí příkazu `nfb-eth -e 1` (popsán v podsekci 2.6.2) je vypnut jeden nebo oba tyto režimy a je testováno, zda pakety:

1. při vypnutém režimu TXMAC se nezobrazí v žádném z čítačů,
2. při vypnutém režimu RXMAC se zobrazí v čítači pro TXMAC, ale nezobrazí se v čítači pro RXMAC.

Toto chování bylo v průběhu implementace práce upravováno a v různých designech bylo různé. Po dohodě s odborným konzultantem bylo chování dle bodů výše prohlášeno za správné a v testovací sadě bylo implementováno jako očekávané.

Poslední částí testování byla funkcionalita etherStats. Jedná se o výpis statistik z karty příkazem `nfb-eth -Sj`, popsaný v sekci 2.6.5. Na základě výpisu byly implementovány 2 testovací funkce: `test_etherstats` a `test_etherstats_conf_mtu`. První jmenovaná se zaměřuje na všechny v současné době testovatelné statistiky kromě těch, které začínají prefixem `conf_`. Ty jsou testovány druhou jmenovanou funkcí.

4.5 Společné části kódu

Některé funkce byly použity v obou částech implementace a proto byly vyčleněny do společného pomocného modulu. Tím je soubor `pytest/src/functional/helper.py`, ve kterém je implementace funkce `get_eth_and_dma_chans`. Jak už její název napovídá, funkce slouží k prostému účelu: zjištění konfigurace karty, konkrétně, jaké DMA kanály a ethernetové porty jsou v dané konfiguraci a pro dané aplikační jádro dostupné. Jejich počty a indexy totiž mohou být u každého designu — resp. každého aplikačního jádra — jiné. Funkce využívá třídu `NdkCardInfo`, ze které lze tuto informaci vyčíst, konkrétně z jejího atributu `dt_ports`.

4.6 Úpravy v testovacím prostředí

Jako poslední stojí za zmínku několik úprav v testovacím prostředí. V odevzdaných zdrojových kódech jsou tyto úpravy demonstrovány formou patch-file, jelikož kompletní zdrojové kódy jsou duševním vlastnictvím sdružení CESNET a nelze je proto šířit dále.

[7] V tomto kontextu (linkové vrstvy) jsou pojmy paket a rámec zaměnitelné.

Kapitola 5

Ověření funkčnosti

U obou testovacích sad byla v průběhu implementace i při následné integraci ověřována jejich funkčnost. Tato byla prováděna opakovaným a pravidelným spouštěním testovacích sad při vývoji na serverech projektu Liberouter:

- **v první fázi**, zejména při implementaci první sady testů, byl použit server `palava`,
- **v druhé fázi** byl kvůli vytíženosti serverů proveden přesun vývoje na server `cider`.

Tyto servery nejsou a v průběhu práce nebyly autorovi práce fyzicky přístupné a proto byl celý vývoj a následné ověřování funkčnosti realizováno pomocí vzdáleného připojení přes již zmíněný protokol SSH, kde byl pro účely vývoje naklonován repozitář ze vzdáleného úložiště prostřednictvím verzovacího nástroje Git. Tento nástroj byl použit také při vývoji a integraci implementovaného kódu do zbytku testovacího prostředí.

Před a v průběhu implementace byly také po diskuzi s odborným konzultantem zvoleny konkrétní síťové karty poskytující (společně s firmwarem) testovanou funkcionalitu:

- **v první fázi** byly použity síťové karty modelů **NFB-200G2QL** a **FB2CGHH** (označované také jako *Tivoli*), připojené k serveru `palava`,
- **v druhé fázi** byla opět modelová řada **NFB-200G2QL**, avšak v tomto případě její další kus, připojený k serveru `cider`,
- po integraci byly testovací sady spouštěny a testovány na síťových kartách **N6010** a **AGI-FH400G** připojených k serveru `mourvedre`.

Výše zmíněný popis platí pro naprostou většinu případů, kdy byla testována funkčnost, vzhledem k univerzální povaze sad lze však tyto využít pro libovolnou kompatibilní síťovou kartu NDK s patřičným firmware a server s patřičně připraveným testovacím prostředím, což bylo v průběhu implementace několikrát vyzkoušeno.

Při každém takovém kontrolním spuštění nad zvolenými síťovými kartami bylo provedeno vyhodnocení pomocí vizuální kontroly ladicích výpisů a také výsledků jednotlivých testovaných případů. Zvláštní pozornost byla věnována takovým případům, u kterých byl negativní výsledek, tedy takovým, u kterých se očekávaný výstup lišil od skutečného.

5.1 Vyhodnocení ladicích výpisů

Nástroje dostupné v testovacím prostředí již v základu implementují mnoho ladicích výpisů. Tyto v průběhu kolekce a přípravy testů (fáze Arrange ze sekce 3.2) poskytují informace o testovaném zařízení, zvoleným režimům a další důležité informace.

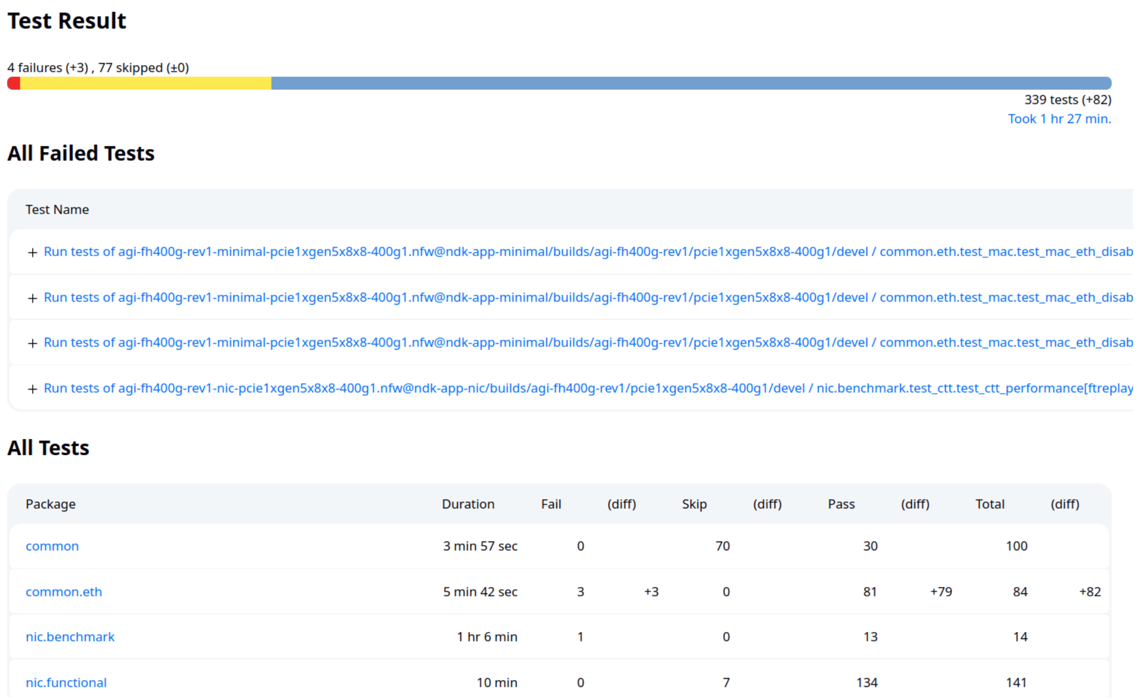
```
21:17:01 INFO common.eth.test_mac: Chosen tested dma index for eth chan 1: 18
21:17:01 DEBUG NdkCardIO: Executing: nfb-eth -d /dev/nfb/by-pci-slot/0000:3b:00.0 -r -L 1522
21:17:01 INFO common.eth.test_mac: Chosen random count of packets to send: 5495
21:17:01 DEBUG NdkCardIO: Executing: nfb-eth -d /dev/nfb/by-pci-slot/0000:3b:00.0 -i 1 -q rx_link
21:17:01 DEBUG src.common.helper: Link status: DOWN
21:17:02 DEBUG NdkCardIO: Executing: nfb-eth -d /dev/nfb/by-pci-slot/0000:3b:00.0 -i 1 -q rx_link
21:17:02 DEBUG src.common.helper: Link status: UP
21:17:02 DEBUG common.eth.test_mac: Non-multicast MAC: 68:e2:3a:a0:d1:a0
21:17:03 DEBUG NdkCardIO: Executing: ndp-transmit -d /dev/nfb/by-pci-slot/0000:3b:00.0 -f /tmp/pytest-of-x
21:17:04 DEBUG NdkCardIO: Executing: nfb-eth -d /dev/nfb/by-pci-slot/0000:3b:00.0 -i 1 -rPv
21:17:04 DEBUG common.eth.test_mac: rx stats: {"bercntr": 1732263, "biperrs": [0, 0, 0, 0, 0, 0, 0, 0, 0,
21:17:04 DEBUG NdkCardIO: Executing: nfb-eth -d /dev/nfb/by-pci-slot/0000:3b:00.0 -jS -i 1
21:17:04 DEBUG NdkCardIO: EtherStats: {
  "octets": 8335915,
  "pkts": 5495,
  "broadcast": 0,
  "multicast": 0,
  "crc_align_errors": 0,
  "undersize": 0,
  "oversize": 0,
  "fragments": 0,
  "jabbers": 0,
  "pkts64": 0,
  "pkts65to127": 0,
  "pkts128to255": 0,
  "pkts256to511": 0,
  "pkts512to1023": 0,
  "pkts1024to1518": 5495,
  "conf_undersize": 0,
  "conf_oversize": 0
}
21:17:04 DEBUG common.eth.test_mac: Checking that octets matches 8335915
21:17:04 DEBUG common.eth.test_mac: Checking that pkts matches 5495
21:17:04 DEBUG common.eth.test_mac: Checking that broadcast matches 0
21:17:04 DEBUG common.eth.test_mac: Checking that multicast matches 0
21:17:04 DEBUG common.eth.test_mac: Checking that crc_align_errors matches 0
21:17:04 DEBUG common.eth.test_mac: Checking that undersize matches 0
21:17:04 DEBUG common.eth.test_mac: Checking that oversize matches 0
21:17:04 DEBUG common.eth.test_mac: Checking that fragments matches 0
21:17:04 DEBUG common.eth.test_mac: Checking that jabbers matches 0
21:17:04 DEBUG common.eth.test_mac: Checking that pkts64 matches 0
21:17:04 DEBUG common.eth.test_mac: Checking that pkts65to127 matches 0
21:17:04 DEBUG common.eth.test_mac: Checking that pkts128to255 matches 0
21:17:04 DEBUG common.eth.test_mac: Checking that pkts256to511 matches 0
21:17:04 DEBUG common.eth.test_mac: Checking that pkts512to1023 matches 0
21:17:04 DEBUG common.eth.test_mac: Checking that pkts1024to1518 matches 5495
21:17:04 DEBUG common.eth.test_mac: Checking that conf_undersize matches 0
21:17:04 DEBUG common.eth.test_mac: Checking that conf_oversize matches 0
PASSED
```

Obrázek 5.1: Ukázka ladicích výpisů testů

Nejinak je tomu i u implementovaných testovacích sad, které před přípravou testu, během něj i po něm informují o vybraných hodnotách v rámci randomizace testovacích parametrů, obsahu porovnávaných výstupů a také informací o aktuálně prováděných akcích. Ukázka ladicích výpisů je vidět na snímku obrazovky 5.1. Tyto ladicí výpisy byly kromě samotných výsledků testů při spuštění vždy z velké části zkontrolovány a porovnány s očekávaným chováním.

5.2 Integrace

Tato část není zadáním přímo vyžadována, nicméně byla u většiny testovacích sad provedena. Jedná se o integraci implementovaného kódu do zbytku testovacího prostředí, kdy byl v několika fázích vývoje kód začleněn do hlavní vývojové větve repozitáře `hwtests`. K tomu byl využit verzovací nástroj Git, prostřednictvím kterého byla vždy vytvořena nová větev pro vývoj a pro tu byl po dokončení dané fáze vytvořen tzv. *Merge request* (MR) v privátním úložišti GitLab sdružení CESNET. **Merge request** vždy prošel několika koly připomínkovacího řízení od odborného konzultanta, přičemž všechny připomínky byly do implementace následně zapracovány. Kromě poslední části druhé testovací sady (funkcionalita `etherStats`) bylo řešení začleněno a je již nasazeno ve vývojovém prostředí, kde je využito při regresním testování. To probíhá automatizovaně každou noc na serverech sdružení nad různými typy designů u různých síťových karet. U poslední části druhé testovací sady je ještě před její integrací nutno počkat na začlenění podpory `etherStats` u příkazu `nfb-eth` do hlavní vývojové větve frameworku NFB.



Obrázek 5.2: Ukázka výsledků testů v CI nástroji Jenkins

Na obrázku 5.2 lze vidět snímek obrazovky z nástroje pro automatickou integraci kódu zvaného Jenkins. Snímek zachycuje výsledky automatizovaných testů síťové karty AGI-FH400G na serveru `mourvedre` ze dne 6. května 2025. Tyto snímky zahrnují i jiné testovací sady sdružení, čistě autorských je na tomto snímku **133** testů ze skupiny **nic.functional** a **82** testů ze skupiny **common.eth**. Lze si všimnout, že v druhé jmenované skupině jsou zde případy, které skončily neúspěchem (Tabulka **All Failed Tests**), protože v době integrace bylo specifikace chování designu změněna a design ještě nebyl na tuto novou specifikaci patřičně adaptován. Jinými slovy, implementované testovací sady odhalily nesrovnalost vůči specifikaci.

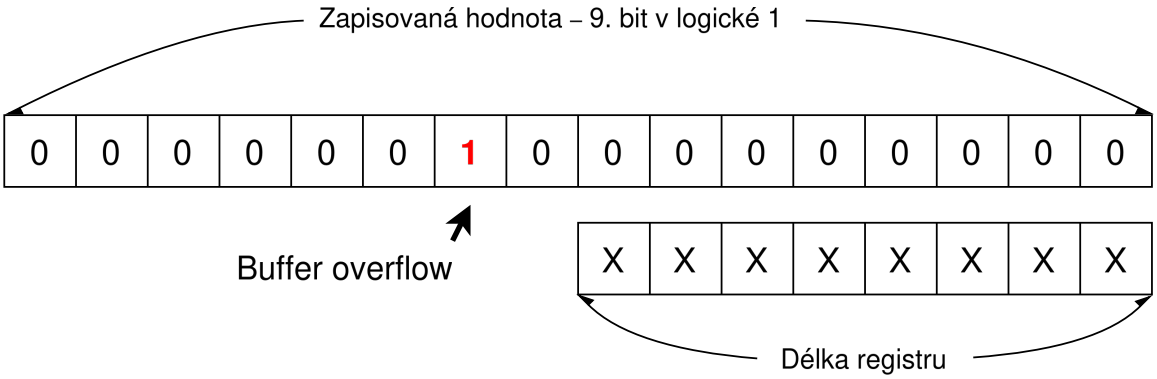
Kapitola 6

Výsledky a možná rozšíření

Při kontrole testovaných výstupů byly odhaleny dvě chyby týkající se nástroje `ndk-nic-ctl` (popsaného v sekci 2.7). Tyto chyby byly po jejich nalezení prostřednictvím odborného konzultanta nahlášeny a později opraveny autorem nástroje, přičemž tato oprava byla znovu testována, mimo jiné i rozšířením testovacích sad o nové případy testování. Také byla nalezena jedna chyba týkající se firmwaru a komponenty MAC, ta však zatím opravena nebyla.

6.1 Chyby v nástroji `ndk-nic-ctl`

Jedna z chyb v nástroji `ndk-nic-ctl` byla objevena při implementaci a ověřování funkčnosti testů funkcionality `mark`. Při implementaci byl učiněn závěr, že ve všech verzích designu pro libovolnou kartu lze v pravidlu adresovat libovolný z 16 dostupných bitů, jak je tomu ostatně uzpůsobena struktura s NIC metadaty. Tato úvaha však byla chybná, protože současná implementace v designu pracuje pouze s jejich dolní polovinou, tedy 8 bity. Takové chování však nebylo zdokumentováno. Pokus o zápis do bitů vyšších vyústil v přetečení[1] (znázorněno na obrázku 6.1) a do daného registru karty tak nástroj efektivně zapsal nulovou hodnotu. Nulová hodnota ale znamená, že konkrétní paket má být místo pouhého označení jednoduše zahozen.



Obrázek 6.1: Přetečení pole při zápisu mark bitu

[1] buffer overflow

Toto chování nebylo pozorováno vždy, jelikož díky využití randomizace nastal tento jev jen ve zhruba 50 % případů, kdy hodnota parametru testu padla do intervalu vyšších osmi bitů (8-15).

Po konzultaci s odborným konzultantem a autorem nástroje `ndk-nic-ctl` bylo toto chování vyhodnoceno jako chyba. Jednalo se zejména o úplnou absenci kontroly správnosti daného bitového indexu. Tato kontrola byla později přidána, konkrétně ve verzi 0.8.0 sady knihoven `pynfb`. Pro ověření správnosti implementované kontroly byl přidán doplňkový test, který kontroluje právě hodnotu mark mimo rozsah (funkce `test_nic_mark_invalid` v souboru `test_nic_filtering.py`).

6.1.1 Pád nástroje při nevalidní struktuře dat

Funkčnost implementace 1. testovací sady stojí především na správném generování konfiguračního souboru pro NIC filtr. Při implementaci však nastaly případy, kdy tato logika nefungovala přesně podle očekávání a při níž byl vygenerován výstup, který nebyl sémanticky či syntakticky správně. V těchto případech bylo očekáváné chování nástroje konfigurační soubor zamítnout s tím, že neobsahuje validní strukturu dat. V nástroji však nebyly správně ošetřeny některé cesty, což místo zamítnutí konfigurace vedlo k pádu nástroje. Toto chování z pochopitelných důvodů nebylo žádoucí, proto bylo v následujících verzích opraveno [31].

6.2 Chybně interpretované sekce v etherStats

Poslední z nalezených chyb byla objevena v ukládání statistik v aplikaci *Minimal*. Ukázalo se, že čítače počtu paketů pro skupiny *broadcast* a *multicast* byly nesprávné, resp. byly mezi sebou vyměněny; při zpracování rámců patřících dle cílové MAC adresy pod multicastový provoz byl navyšován čítač broadcastových rámců a naopak. Tato chyba se z testovaných aplikací projevovala pouze u *Minimal*, aplikace *NIC* měla tuto funkcionalitu v pořádku, resp. podobná chyba se u ní neprokázala a výsledek testů byl pozitivní. Chyba byla opět nahlášena, do uzávěrky textové části práce však nebyla opravena [31].

6.3 Možná další rozšíření

Při návrhu testovacích sad bylo nalezeno několik oblastí, kterými by se dal vývoj testovacích sad dále ubírat. Některé z nich vyžadují implementaci nového API nebo rozsáhlejší změny v testovacím prostředí, aby mohly být důkladně otestovány, proto jim nebyla věnována přílišná pozornost.

6.3.1 Rozšíření 1. sady

V první testovací sadě — navržené pro NIC filtr — by bylo možné dále testovat například kapacitu filtru, kdy tento může pojmout v konfiguraci najednou až 128 pravidel. Tento scénář stále spadá do oblasti funkčních a nikoliv benchmark testů, nicméně testování této funkcionality by vyžadovalo výrobu poměrně rozsáhlého vzorku dat — bylo by nutno naplnit tabulku 128 pravidly a pak nejlépe u každého testovat, zda se opravdu chová dle předpokladů a nedochází například k překryvu spouštěných akcí.

Dále v testovací sadě pro NIC filtr chybí testování filtrace na základě VLAN, což může být nepřiliš častý případ užití, nicméně bylo by vhodné jej testovat také.

6.3.2 Rozšíření 2. sady

Také druhá testovací sada skýtá další možnosti, kudy by mohl být její vývoj dále veden. Za zmínku stojí zejména testování nevalidních paketů padajících do sekcí `Erroneous` (počty chybných paketů), resp. jabbers[2] v kontextu `etherStats`.

Nástroje v testovacím prostředí neumožňují generování a vysílání takových paketů, buď kvůli omezením knihoven (Scapy, sekce 2.8.3) nebo i s účelem nemožnosti generování škodlivého provozu. Pro tyto účely lze ve sdružení CESNET použít modulární zátěžový generátor paketů od firmy Spirent Communications[3].

Tyto generátory jsou v CESNETu připojeny jen na některé síťové karty u konkrétních serverů. Pro účel implementace testů by bylo nutné zvolit vhodné kandidáty pro obě tyto skupiny. Dále by bylo nutné doimplementovat ty části stávajícího API v testovacím prostředí, které by byly zodpovědné za správné nastavení generátoru, zejména nastavení podvržení zmíněného kontrolního součtu.

[2] Jabbers jsou v kontextu RFC 1271 pakety delší než 1518 oktetů

[3] Firma Spirent se specializuje na vývoj HW i SW řešení pro střední a velké sítě. Její testovací hardware lze zhlédnout například zde: https://www.spirent.com/products/testcenter-hardware.

Kapitola 7

Závěr

Cílem této práce byla implementace nových testovacích sad pro síťové karty sdružení CESNET do stávajícího testovacího prostředí. Nejprve proběhlo seznámení s testovacím prostředím pro síťové karty vyvíjené sdružením CESNET a v něm používaným frameworkem `pytest`. Následně mělo být s pomocí nastudované problematiky učiněno rozšíření o nové testovací sady pro funkcionalitu, která ještě není pokryta žádným z dosud implementovaných automatických testů. Implementačním výstupem mé práce jsou dvě testovací sady implementující funkcionální testování konkrétních vlastností firmwaru síťových karet. První sada a větší část druhé sady je již integrována do nástrojů pro automatickou integraci. V případě poslední části druhé testovací sady, která se zaměřuje na funkcionalitu `etherStats`, je vyčkáváno na začlenění testované funkcionality a testovací sada je zatím plně připravena na integraci do stávajícího řešení. Většina implementované funkcionality je tak již nasazena v ostrém provozu, v nástrojích pro automatickou integraci kódu běžících ve sdružení CESNET. U nalezených chyb bylo vyhodnoceno, zda se jedná o chyby testů, testovacího prostředí, nástrojů či samotného firmwaru karet. Byly nalezeny celkem tři podstatné chyby — dvě v nástrojích pro práci s kartami a jedna v samotném firmwaru. První dvě chyby byly již opraveny a implementované sady nyní plní převážně roli regresních testů, které jsou v pravidelných intervalech automatizovaně spouštěny nad vybranými designy firmwaru pro síťové karty. Na závěr jsou analyzována a popsána další vhodná rozšíření, kterými by vývoj této práce mohl dále pokračovat, zejména testování VLAN u NIC filtru a FCS u komponenty MAC.

Literatura

[1] Bray, T. *The JavaScript Object Notation (JSON) Data Interchange Format* RFC 8259. RFC Editor, prosinec 2017. Dostupné z: https://doi.org/10.17487/RFC8259. [cit. 22. dubna 2025].

[2] Bulić, P. Direct Memory Access. In: *Understanding Computer Organization: A Guide to Principles Across RISC-V, ARM Cortex, and Intel Architectures*. Cham: Springer International Publishing, 2024, s. 155–176. ISBN 978-3-031-58075-8. Dostupné z: https://doi.org/10.1007/978-3-031-58075-8_3.

[3] CESNET, z.s.p.o. *CESNET vyvinul ve spolupráci s francouzskou společností Reflex CES akcelerační kartu pro 400Gb sítě* online. 2024. Dostupné z: https://www.cesnet.cz/o-nas/tiskove-zpravy-1/cesnet-vyvinul-ve-spolupraci-s-francouzskou-spolecnosti-reflex-ces-akceleracni-kartu-pro-400gb-site-34. [cit. 27. prosince 2024].

[4] CESNET, z.s.p.o. *NDK FPGA Development* online. 2024. Dostupné z: https://cesnet.github.io/ndk-fpga/devel/index.html. [cit. 16. ledna 2025]. GitHub repozitář sdružení CESNET.

[5] CESNET, z.s.p.o. *NDK FPGA Development* online. 2024. Dostupné z: https://cesnet.github.io/ndk-fpga/devel/ndk_core/doc/readme.html#ndk-architecture. [cit. 16. ledna 2025]. GitHub repozitář sdružení CESNET.

[6] CESNET, z.s.p.o. *NDK Software Development* online. 2024. Dostupné z: https://cesnet.github.io/ndk-sw/index.html. [cit. 16. ledna 2025]. GitHub repozitář sdružení CESNET.

[7] CESNET, z.s.p.o. *Síť CESNET3* online. 2024. Dostupné z: https://www.cesnet.cz/sit-cesnet3. [cit. 18. prosince 2024].

[8] Cisco Networking Academy. *Introduction to Networks: Modules 4, 6, 7* online. 2025. Dostupné z: https://netacad.fit.vutbr.cz/wp-content/uploads/ccna/itn/lectures/p2-m4-m6-m7.pdf. [cit. 19. dubna 2025]. Studijní materiály Cisco Networking Academy.

[9] GitHub, Inc.. *Using a Matrix Strategy in GitHub Actions Workflows* online. 2025. Dostupné z: https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/running-variations-of-jobs-in-a-workflow#using-a-matrix-strategy. [cit. 23. února 2025]. Online dokumentace.

[10] Harris, G. a Richardson, M. *PCAP Capture File Format*. Internet-Draft draft-ietf-opsawg-pcap-05. Internet Engineering Task Force, březen 2025. Dostupné z: https://datatracker.ietf.org/doc/draft-ietf-opsawg-pcap/05/. [cit. 25. dubna 2025]. Work in Progress.

[11] Krekel, H. a pytest-dev team. *Pytest: Framework pro testování v jazyce Python* online. 2015. Dostupné z: https://docs.pytest.org/en/stable/. [cit. 21. prosince 2024]. Online dokumentace.

[12] Kubálek, J.; Cabal, J.; Špinler, M. a Iša, R. DMA Medusa: A Vendor-Independent FPGA-Based Architecture for 400 Gbps DMA Transfers. In: Editors, I., ed. *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2021, s. 258–258. ISBN 978-1-6654-3555-0.

[13] Liberouter Project. *Liberouter: High-performance networking and monitoring solutions* online. 2024. Dostupné z: https://www.liberouter.org/. [cit. 27. prosince 2024].

[14] Liberouter Project. *NDK Hardware Tests* online. 2024. Dostupné z: https://ndk.gitlab.liberouter.org/ndk/hwtests. [cit. 15. ledna 2025]. Privátní instance úložiště GitLab dostupná online přes Liberouter VPN.

[15] Liberouter Project. *NDK-APP-NIC documentation* online. 2025. Dostupné z: https://ndk.gitlab.liberouter.org:5051/ndk-app-nic/index.html. [cit. 15. ledna 2025]. Privátní instance úložiště GitLab dostupná online přes Liberouter VPN.

[16] Liberouter Project. *NDK: Network Development Kit* online. 2025. Dostupné z: https://www.liberouter.org/ndk/. [cit. 6. dubna 2025]. Oficiální stránka projektu NDK.

[17] Liberouter Project. *NDK-NIC-CTL tool* online. 2025. Dostupné z: https://ndk.gitlab.liberouter.org:5051/ndk-app-nic/app/doc/sw_tool.html. [cit. 15. ledna 2025]. Privátní instance úložiště GitLab dostupná online přes Liberouter VPN.

[18] Luizelli, M.; Vogt, F.; Matos, G.; Cordeiro, W.; Filho, A. et al. SmartNICs: The Next Leap in Networking. In: Rothenberg, C. E., ed. *SmartNICs: The Next Leap in Networking* online. Květen 2024, s. 40–89. ISBN 9788576696087. Dostupné z: https://doi.org/10.5753/sbc.15408.7.2.

[19] Matoušek, P. *Síťové aplikace a správa sítí – Prostředky pro správu sítí*. Vysoké učení technické v Brně, Fakulta informačních technologií, 2024. Dostupné z: https://moodle.vut.cz/mod/resource/view.php?id=450434. [cit. 21. prosince 2024]. Materiály ke kurzu ISA dostupné studentům kurzu.

[20] pytest Development Team. *Command-line Flags in pytest* online. 2025. Dostupné z: https://docs.pytest.org/en/stable/reference/reference.html#command-line-flags. [cit. 19. dubna 2025]. Oficiální dokumentace k příkazovým přepínačům v pytest.

[21] pytest Development Team. *Pytest Documentation* online. 2025. Dostupné z: https://docs.pytest.org/en/stable/explanation. [cit. 7. dubna 2025]. Oficiální dokumentace k frameworku pytest.

[22] Python Software Foundation. *PEP 318 – Decorators for Functions and Methods* online. 2004. Dostupné z: https://peps.python.org/pep-0318/. [cit. 23. února 2025]. Python Enhancement Proposal.

[23] Python Software Foundation. *Assert statement* online. 2023. Dostupné z: https://docs.python.org/3/reference/simple_stmts.html#grammar-token-python-grammar-assert_stmt. [cit. 15. ledna 2025]. Online dokumentace.

[24] Python Software Foundation. *Emulating Numeric Types in Python* online. 2025. Dostupné z: https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types. [cit. 9. dubna 2025]. Online dokumentace k emulaci numerických typů v Pythonu.

[25] Python Software Foundation. *Enum — Support for enumerations* online. 2025. Dostupné z: https://docs.python.org/3/library/enum.html. [cit. 19. dubna 2025]. Online dokumentace k modulu enum v Pythonu.

[26] Python Software Foundation. *Python Glossary* online. 2025. Dostupné z: https://docs.python.org/3/glossary.html. [cit. 23. února 2025]. Online dokumentace.

[27] PyYAML Project. *PyYAML Documentation* online. 2025. Dostupné z: https://pyyaml.org/wiki/PyYAMLDocumentation. [cit. 6. dubna 2025]. Oficiální dokumentace pro PyYAML.

[28] Scapy Project. *Scapy Documentation* online. 2025. Dostupné z: https://scapy.readthedocs.io/en/latest/. [cit. 6. dubna 2025]. Oficiální dokumentace pro Scapy.

[29] SSH Communications Security. *What is SSH (Secure Shell)?* online. 2025. Dostupné z: https://www.ssh.com/academy/ssh. [cit. 19. dubna 2025].

[30] The Linux Kernel Documentation. *NUMA (Non-Uniform Memory Access)* online. 2018. Dostupné z: https://www.kernel.org/doc/html/v4.18/vm/numa.html. [cit. 12. dubna 2025]. Oficiální dokumentace Linuxového jádra k NUMA.

[31] Vodák, D. *Soukromá konverzace s Ing. Davidem Vodákem ze sdružení CESNET na platformě Slack*. 2025.

[32] Waldbusser, S. *Remote Network Monitoring Management Information Base* RFC 1271. RFC Editor, listopad 1991. Dostupné z: https://doi.org/10.17487/RFC1271. [cit. 22. dubna 2025].

[33] Waldbusser, S. *Remote Network Monitoring Management Information Base* RFC 2819. RFC Editor, květen 2000. Dostupné z: https://doi.org/10.17487/RFC2819. [cit. 22. dubna 2025].

[34] YAML Working Group. *YAML Ain't Markup Language (YAML™) Version 1.2.2 Specification* online. 2021. Dostupné z: https://yaml.org/spec/1.2.2/. [cit. 19. března 2025]. Oficiální specifikace jazyka YAML.

[35] Čejka, T. *Quo vadis, ipfixprobe?* online. 2023. Dostupné z: https://www.youtube.com/watch?v=0Y4Msg0zm4A. [cit. 21. prosince 2024]. Přednáška na konferenci LinuxDays 2023.

[36] Špinler, M. *Ukázka JSON výstupu: nfb-eth -j.* 2025. Soukromý soubor sdílený přes Slack.