Univerzita Pardubice
Fakulta elektrotechniky a informatiky

GUI editor pro Mininet

Bc. Tomáš Vyčítal

Diplomová práce

2019

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2018/2019**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Tomáš Vyčítal**
Osobní číslo:	**I17228**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**GUI editor pro Mininet**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

Cílem diplomové práce je návrh a realizace grafického editoru pro emulátor Mininet. Při návrhu bude napodobena funkcionalita aplikace MiniEdit. Teoretická část bude složená z popisu prostředí emulátoru Mininet a jeho komponet a z ukázkových úloh, které budou realizovány v navrženém GUI editoru.

Rozsah grafických prací: **10**

Rozsah pracovní zprávy: **cca 40–50 stran**

Forma zpracování diplomové práce: **tištěná**

Seznam odborné literatury:

NADEAU, Thomas D. a Kenneth GRAY. SDN: software defined networks. Beijing: O'Reilly, 2013. ISBN 978-1-449-34230-2. SUMMERFIELD, Mark. Python 3: výukový kurz. Brno: Computer Press, 2010. ISBN 9788025127377. Mininet Walkthrough - Mininet [online]. 2017 [cit. 2018-10-04]. Dostupné z: http://mininet.org/walkthrough/.

Vedoucí diplomové práce: **Ing. Miroslav Dvořák, Dipl.tech.**
Katedra informačních technologií

Datum zadání diplomové práce: **22. října 2018**

Termín odevzdání diplomové práce: **18. května 2019**

Univerzita Pardubice Česká republika

L.S.

Ing. Zdeněk Němec, Ph.D.
děkan

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 17. listopadu 2018

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 18. 5. 2019

Bc. Tomáš Vyčítal

Poděkování

Děkuji Ing. Miroslavu Dvořákovi, Dipl.tech. za vedení a připomínky při vývoji aplikace a také za cenné rady k vypracování textové části práce.

Anotace

Hlavní náplní této diplomové práce je naprogramování SPA Mininet editor, ten umožňuje vytvoření a základní nastavení topologie SDN v grafickém prostředí. Funkcionalita zahrnuje import a export ve formě kompletního projektu nebo skriptu pro emulaci v Mininetu, pak také export obrázku topologie a adresního plánu. Práce dále obsahuje stručný úvod do počítačových sítí včetně SDN, Mininetu a ukázkové úlohy z této oblasti s využitím Mininet editoru pro jejich vypracování.

Klíčová slova

GUI, JavaScript, Mininet, PWA, SDN, SPA, Vis, Vue

Title

GUI editor for Mininet

Annotation

The main output of this thesis is Mininet editor, an SPA for creating and configuring SDN topologies in GUI. Features include import and export of entire project and a script that can be used to emulate the topology in Mininet. It is also possible to export an image of the topology and an addressing plan. A part of this thesis is also devoted to a brief introduction to computer networks including SDN, Mininet and example assignments that can be completed using Mininet Editor.

Keywords

GUI, JavaScript, Mininet, PWA, SDN, SPA, Vis, Vue

Obsah

Seznam obrázků

Seznam tabulek

Seznam fragmentů kódu

Seznam zkratek

ANTLR	Another Tool For Language Recognition
API	Application Programming Interface
CFS	Completely Fair Scheduler
CLI	Command-Line Interface
CPU	Central Processing Unit
DDoS	Distributed Denial of Service
DOM	Document Object Model
GNU	GNU's Not Unix!
GUI	Graphical User Interface
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
LTE	Long-Term Evolution
MAC	Media Access Control
ONF	Open Networking Foundation
OVS	Open vSwitch
PDF	Portable Document Format
PNG	Portable Network Graphics
PWA	Progressive Web Application
QoS	Quality of Service
RAM	Random-Access Memory
SDN	Software Defined Network
SPA	Single Page Application
TC	Traffic Control
UUID	Universally Unique Identifier
VCS	Version Control System
VIRL	Virtual Internet Routing Lab
WAN	Wide-Area Network

Úvod

Cílem teoretické části je stručné seznámení se s tradičními sítěmi a softwarově definovanými sítěmi, s důvodem jejich vzniku a s jejich výhodami. Dalším cílem je poskytnutí stručného přehledu existujících simulátoru a emulátorů softwarově definovaných sítí včetně Mininetu, kterému bude věnována samostatná kapitola s podrobnějším popisem a návodem k použití.

Hlavním cílem je návrh a naprogramování aplikace Mininet editor. Aplikace bude napodobovat funkcionalitu již existujícího MiniEditu, ta bude nicméně podstatně rozšířena a Mininet editor bude celkově propracovanější a uživatelsky přívětivější.

V poslední části budou vypracovány ukázkové úlohy z oblasti softwarově definovaných sítí. Ty bude možné vypracovat právě s pomocí Mininet editoru, vyexportovat je a následně ověřit správnou funkčnost při jejich emulaci.

1 Tradiční sítě

V tradičním pojetí síťové infrastruktury je síť složena ze switchů, routerů a dalších zařízení, která pracují do značné míry nezávisle na sobě. Jejich chování je koordinováno pomocí vhodné konfigurace každého zařízení. Konfigurace velkého množství zařízení v síti, zejména pokud jsou používána zařízení od různých výrobců, je komplikovaná záležitost náchylná na chyby správce sítě, obzvláště pokud jich je více. Dále se používají různé jednoúčelové protokoly, které zajišťují například automatickou distribuci a tvorbu směrovacích pravidel, agregaci linek nebo odstranění smyček v topologii. Tyto protokoly jsou často distribuované, což má za následek, že stráví určité množství času zjišťováním topologie nebo změn v ní a domlouváním se na dalších činnostech (například na tom, které spoje se budou používat nebo který prvek bude hlavní, který převezme funkci hlavního při jeho výpadku). Dalším problémem jsou proprietární protokoly, které nespolupracují s produkty jiných výrobců (vendor lock-in), a pomalá standardizace a adopce nových (mj. otevřených) protokolů a myšlenek.

V terminologie SDN má tedy každé zařízení v datové vrstvě sítě malý kousek kontrolní vrstvy, který obsahuje část řídící logiky a rozhoduje o přeposílání paketů v síti. Z výše zmíněného vyplývá, že kontrolní vrstva, je roztříštěná na velkém množství zařízení v datové vrstvě, která ale nemají rozsáhlé znalosti o topologii celé sítě a typicky ani nijak vysoký výpočetní výkon. Kvůli tomu nemohou rychle a dostatečně inteligentně reagovat na změny v síti anebo v provozu na síti. Mají jen omezenou schopnost využívat dostupnou infrastrukturu a vyžaduji velké rezervy pro zajištění rozumné propustnosti a přijatelné QoS za všech okolností.

2 Softwarově definované sítě

Základní myšlenkou softwarově definovaných sítí je oddělení datové vrstvy, která zajišťuje samotný přenos dat, od vrstvy kontrolní, která rozhoduje o tom, kam, kudy, jak a jestli vůbec se data budou v síti přeposílat.

Kontrolní vrstva je nezávislá na prvcích datové vrstvy, které spravuje, což ji dává znalosti o celé topologii nebo o její velké části. Veškerá logika kontrolní vrstvy je umístěna na centralizovaném kontroleru nebo je distribuovaná na více kontrolerech, díky čemuž je možné rozložit zátěž a lépe řešit výpadky jednotlivých kontrolerů. V reálném čase také dostává informace o vytíženosti sítě a poruchách v ní od kontrolovaných zařízení z datové vrstvy. Díky tomu může velmi rychle učinit rozhodnutí pro obnovení propustnosti sítě vhodným rozložením zátěže při poruše v datové vrstvě nebo při neobvykle vysokém vytížení, které může být způsobeno například i DDoS útokem. Značnou výhodou oproti tradičním sítím je zde možnost programování vlastních modulů (v kontrolerech často nazývaných aplikacemi) v kontrolní vrstvě nebo přímo vlastního kontroleru. V těchto modulech lze snadno implementovat libovolné algoritmy, které mohou být ušité na míru dané síti, a poskytovat podstatně lepší řízení dané sítě než obecné algoritmy poskytované výrobci tradičních switchů a routerů.

Datová vrstva se stará jen o přenos dat a o hlášení statistik o aktuálním stavu sítě kontrolerům, které dělají všechna rozhodnutí. Jednotlivé switche na datové vrstvě nijak nerozhodují o provozu v síti, jen dodržují pravidla přijatá od kontroleru. Tato pravidla mohou obsahovat instrukce k přeposílání paketů podle určených kritérií (například podle zdrojové IP adresy, cílové MAC adresy nebo vstupního portu na switchi) na určité porty switche, k provedení jednoduchých úprav na paketech, zahazování paketů atd. Z toho plyne, že v SDN nedávají velký smysl routery (s výjimkou routerů na hranici sítě), jelikož směrování jsou schopné provádět switche na základě instrukcí od kontrolerů.

Komunikace mezi kontrolní a datovou vrstvou se realizuje pomocí specializovaného protokolu, nejběžnějším z nich je OpenFlow, ale existují i další. Jednou variantou realizace samotného přenosu je samostatná síťová infrastruktura (out-of-band), která propojuje switche a kontrolery pomocí fyzických síťových spojů mimo SDN. Druhou variantou je přenos v rámci datové vrstvy SDN (in-band) pomocí logických spojů, taková síť pak ale musí zajistit konektivitu k alespoň jednomu kontroleru pro prvotní spuštění

sítě a také pro opětovné obnovení funkčnosti po poruše. Na druhé straně ale dokáže využít již existující infrastrukturu, a tím pádem snížit náklady na ni, protože nevyžaduje paralelní síť pro propojení switchů s kontrolery.

Možnosti využití softwarově definovaných sítí jsou teoreticky stejně široké jako využití tradičních sítí (každou tradiční síť lze konvertovat na SDN). V některých případech ale využití SDN nedává velký smysl (například pro typickou domácnost nebo malou kancelář). Hlavní využití SDN nachází v oblastech, kde je potřeba přenášet velké množství dat a spravovat větší množství infrastruktury, typicky se jedná o datacentra a sítě WAN. Tyto sítě pak mohou lépe rozkládat zátěž ve své infrastruktuře a dynamicky reagovat na měnící se požadavky uživatelů, na poruchy infrastruktury, útoky na ni, zajišťovat QoS atd. Také mohou dle potřeby vypínat části infrastruktury (včetně serverů nebo celých částí datacenter), které nejsou v dané době využívané nebo jsou málo vytížené a jejich činnost může převzít jiná část infrastruktury, a tím značně snížit spotřebu energie. Využití SDN také značně usnadňuje oddělení komunikace různých entit na stejné fyzické infrastruktuře, tedy vytvoření soukromých „Internetů“ například pro firmy nebo státní instituce, v porovnání s tradičními sítěmi jsou také podstatně snazší jakékoli změny v těchto politikách.

2.1 OpenFlow

Klíčovou částí SDN je southbound API, tedy rozhraní přes které probíhá komunikace mezi datovou a kontrolní vrstvou, v této oblasti v současné době dominuje protokol OpenFlow. Zajišťuje komunikaci mezi kontrolery na kontrolní vrstvě a switchi na datové vrstvě.

Komunikace začíná objevením jednotlivých zařízení a následně dohodnutím verze protokolu (zvolí se ta nejvyšší, kterou ovládají obě zařízení), k tomu slouží hello zprávy. Poté si kontrolery zjistí schopnosti jednotlivých switchů (zprávy features request a response) a provedou základní konfiguraci (zpráva set config). Další akce kontrolerů spočívají v instalaci flow pravidel (zprávy add, delete a modify) a sběru dat ze switchů (zprávy multipart request). Mimo toho si kontrolery ještě zjišťují selhání jednotlivých switchů (pomocí periodického zasílání hello zpráv). Na základě těchto dat pak kontroler vyhodnocuje situaci a upravuje flow pravidla na switchích. [1]

1.0 Původní verze z roku 2009. Umožňuje kontroleru nastavit ve switchi jednu tabulku, kde má každé pravidlo přiřazenu jednu akci. Tato varianta, byť výhodná z hlediska požadavků na hardware, se ukázala jako nevyhovující, protože pro některé operace je nutné vytvořit velké množství pravidel.

1.1 Tato verze přinesla mimo jiného podporu pro skupinové tabulky, což značně zvýšilo nároky na hardware, ale zároveň velmi zefektivnilo práci s pravidly. Díky více tabulkám lze některé operace řešit ve více krocích místo obrovského množství pravidel, které by bylo potřeba v předchozí verzi protokolu. Také přibyly skupinové tabulky, které umožňují například rozkládání zátěže mezi více alternativních cest anebo rychlé přejití na záložní cestu v případě výpadku té hlavní bez čekání na instrukce od kontroleru.

1.2 S touto verzí přišla podpora pro flexibilní pravidla, která pro tuto i pro následující verze značně usnadnila přidávání nových pravidel a protokolů (jedním z nich je základní podpora IPv6). Důležitým milníkem je také možnost přidat do sítě více kontrolerů, které mohou fungovat současně a sdílet zátěž anebo jako hlavní a záložní kontrolery pro případ poruchy.

1.3 Tato verze zlepšila podporu QoS a také rozšířila možnosti nakládání s pakety, které nevyhoví žádnému pravidlu (dříve je šlo pouze zahodit nebo přeposlat na kontroler).

1.4 V této verzi přibyly synchronizované tabulky, které umožňují, aby se změny stavu v jedné tabulce promítly i do druhé tabulky, v případě obousměrné synchronizace i opačným směrem. Další novinkou je funkce zvaná bundles, jedná se o možnost seskupit sekvenci modifikací jako atomickou jednotku, která se buďto provede celá nebo v případě jakékoli chyby dojte k návratu do stavu před první modifikací ze skupiny.

1.5 Novinkou v této verzi je výstupní tabulka (egress table), která umožňuje zpracovat pakety nejen na vstupu do switche, ale i na jeho výstupu. Dále došlo k obohacení seskupování modifikací (bundles) o časový údaj, podle kterého se switch pokusí modifikace provést, to má za účel zlepšit synchronizaci více switchů.

Budoucnost Velkými problémy protokolu OpenFlow jsou zejména jeho monolitičnost a velké množství volitelných funkcí. Díky monolitické architektuře není snadné protokol dále vyvíjet a rozšiřovat, což vede k zaostávání protokolu za potřebami jeho uživatelů, kteří poté mohou volit konkurenční protokoly nebo kombinace více protokolů. Volitelné funkce způsobují značnou variabilitu mezi jednotlivými výrobci, navíc konkretní implementace se svým chování mohou drobně odlišovat, což může být způsobeno i nedodržením specifikace. Tyto problémy pak mohou způsobovat proprietární uzamčení (také známé jako vendor lock-in) i při používání otevřeného protokolu, zabránění tomuto je nicméně hlavním důvodem existence protokolu OpenFlow. [2]

ONF, která vyvíjí i protokol OpenFlow, ve spolupráci s více než dvěma desítkami dalších společností pracuje na novém projektu s názvem Stratum, ten by měl být modulárním projektem řešícím všechny výše zmíněné problémy. Projekt Stratum v době psaní této práce (25. 2. 2019) ještě nebyl zveřejněn, měl by ale být vydán s otevřeným zdrojovým kódem pod licencí Apache 2.0. [3]

2.2 Simulátory

Výstavba skutečné SDN je finančně i časově náročná a v případě, že je potřeba pouze otestovat určitý koncept, tak i zbytečná. Proto se využívají simulátory, ve kterých je možné si rychle a jednoduše vytvořit topologii a otestovat nad ní různé scénáře. Přičemž se dá vše jednoduše měnit, kopírovat, sdílet atd.

ns-3 Diskrétní událostně orientovaný simulátor počítačových sítí s otevřeným zdrojovým kódem (GNU GPLv2) určený předně pro výzkum a výuku. Podporuje jak IP tak i spoustu dalších protokolů. Z hlediska simulovatelné infrastruktury jsou k dispozici i bezdrátové sítě jako jsou Wi-Fi, WiMAX nebo LTE. Kromě simulace izolované sítě umožňuje i simulaci sítí připojených na a komunikujících se skutečným světem. Ve stádiu testování a vyhodnocování je v současné době (21. 2. 2019) možnost využití skutečných aplikací a jader v rámci simulace. [4]

2.3 Emulátory

Často je výhodné nebo i nezbytné použít software ze skutečného světa při testování určitého scénáře. K tomu se využívá emulace, která umožňuje vyzkoušený scénář snadno nasadit do praxe, protože stejný software, který běžel v rámci testování, lze v nezměněné podobě nebo pouze s drobnými změnami nasadit na skutečnou síť. Nevýhodou jsou nicméně vyšší nároky emulace.

Mininet Tento emulátor je podrobně popsán v kapitole 3.

MaxiNet Umožňuje rozdělit síť Mininetu na více zařízení a stará se o komunikaci mezi nimi. Dále poskytuje API pro správu této distribuované emulace. Jeho další vlastnosti odpovídají Mininetu, na kterém je postaven. [5]

EstiNet Komerční simulátor a emulátor IP sítí. Obsahuje možnost vizualizace simulované sítě včetně animovaného průběhu přenosu paketů sítí. Dále umožňuje propojení s okolním světem a využití libovolných linuxových aplikací v rámci simulace. [6]

VIRL Celým názvem Virtual Internet Routing Lab. Tento nástroj, vyvíjený společností Cisco, umožňuje modelování počítačových sítí s využitím emulace autentických verzí operačních systémů, které jsou dodávány na zařízeních společnosti Cisco. Jedná se o proprietární a komerční software. [7]

3 Mininet

Mininet je emulátor počítačových sítí, který vytváří a spravuje virtuální hosty, switche, kontrolery a spojení mezi nimi na běžném počítači nebo serveru. Také zajišťuje připojení switchů na kontrolery spuštěné mimo Mininet. Na těchto virtuálních zařízeních běží identický software, který by byl provozován na skutečném fyzickém zařízení. Díky tomu si uživatel může otestovat síť tak, jako by ji skutečně fyzicky měl k dispozici, a to často i na obyčejném laptopu (v závislosti na velikosti sítě a výkonnosti laptopu). To vše bez nutnosti nakupovat drahý hardware a fyzicky sestavovat celou síť, což také značně usnadňuje jakékoli změny v dané síti. Virtuální síť Mininetu lze snadno přenášet mezi zařízeními, sdílet mezi více uživately, kteří s ní mohou pracovat zároveň na svých strojích nezávisle na ostatních, nebo ji i verzovat pomocí VCS. K ovládání Mininet obsahuje CLI a Python API pro vybudování, správu, debugování a testování emulované sítě, práci s její topologií a s OpenFlow protokolem.

K dosažení výše zmíněných funkcí používá virtualizaci na úrovni procesů, díky které je možné emulovat tisíce zařízení na jednom počítači (přesný počet záleží na rychlosti CPU, množství RAM atd.) a zároveň zajistit jejich izolaci pro věrné napodobení podmínek skutečného světa. Určitou nevýhodou tohoto přístupu je sdílení jádra systému, z čehož vyplývá, že všechny simulované hosty, switche a kontrolery (pokud daný kontroler neběží na jiném zařízení nebo ve virtuálním stroji) musí běžet na stejném jádru. V současné době je jediným podporovaným jádrem Linux, který je ale tím nejběžnějším jádrem pro SDN síťovou infrastrukturu a je běžný i na serverech, takže v praxi to nezpůsobuje velké problémy. Omezením je také počítač, na kterém Mininet běží, jelikož jeho zdroje nejsou neomezené. Kvůli tomu zařízení, která by se ve skutečnosti neměla nijak ovlivňovat, spolu soupeří o zdroje (například čas na CPU nebo operační paměť), což může značně omezit propustnost celé emulované sítě. [8]

3.1 Použití

Mininet ke svému běhu potřebuje mít nainstalovaný a spuštěný Open vSwitch Daemon, na systému se SystemD jej lze spustit pomocí příkazu `systemctl start ovs-vswitchd`. Hlavním příkazem Mininetu je příkaz `mn`, ten musí být spuštěn s oprávněními uživa-

tele root, obvykle tedy pomocí příkazu `sudo mn`. Hned na začátku je velmi dobré zmínit parametr `-c`, který se postará o úklid po nestandardním ukončení Mininetu. Nejjednodušším způsobem spuštění Mininetu je příkaz `mn` bez argumentů, ten spustí Mininet s jednoduchou topologií (jeden switch řízený jedním kontrolerem se dvěma připojenými hosty). Topologii je také možné určit pomocí parametru `--topo`, ten stačí sám o sobě pro vestavěné topologie (například linear, tree nebo torus), pokud je ale použita vlastní topologie, tak musí být použit v kombinaci s parametrem `--custom` s cestou k souboru, ve kterém je tato topologie definována. Pomocí parametrů `--host`, `--switch` a `--controller` lze určit typ prvků použitých v topologii, například pomocí `--switch ovsbr` lze použít Open vSwitch s takovým nastavením, že v nepřítomnosti kontroleru bude fungovat ve standalone módu.

Po spuštění se ocitnete v prostředí příkazové řádky Mininetu. Příkazy v CLI Mininetu jsou buď globální například `help` pro zobrazení seznamu příkazů s nápovědou, `help nodes` pro nápovědu k příkazu `nodes` nebo `dpctl dump-flows` pro vypsání všech flow pravidel. Další příkazy jsou pak lokální pro jednotlivá zařízení, například pro host h1 `h1 help` pro zobrazení seznamu příkazů a nápovědy nebo `h1 ip a` pro vypsání IP adres. Lze také spouštět libovolné příkazy, které jsou k dispozici v prostředí shellu, pomocí příkazu `sh`, například `sh ovs-ofctl dump-flows s1` pro vypsání flow pravidel ze switche s1 nebo `sh echo Hello World` pro vypsání textu Hello World. Stejným způsobem je zpřístupněno i Python API pomocí příkazů `px`, který vykoná kód v Pythonu, a `py`, který vykoná kód v Pythonu a vypíše jeho výstup do terminálu. Z CLI lze spouštět i grafické aplikace, například příkazem `h1 wireshark-gtk &` lze spustit Wireshark na hostu h1 (bez znaku & na konci by Wireshark až do svého ukončení blokoval CLI). Pro práci v CLI na jednotlivých prvcích je možné si spustit samostatné terminály, například `xterm h1` pro spuštění Xtermu pod hostem h1 nebo `xterm h1 h2 h2` pro spuštění jednoho Xtermu pod hostem h1 a dvou pod hostem h2.

Tabulka 1: Parametry příkazu mn

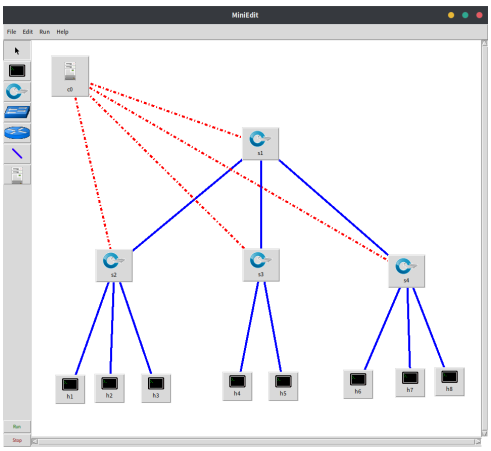
–clean	úklid po nestandardním ukončení Mininetu
–controller remote	použije určený typ kontrolerů (remote se připojí na port 6653 nebo 6633)
–custom path	nahraje váš vlastní kód (například topologii)
–help	nápověda
–host cfs	použije určený typ hostů (cfs podporuje TC a používá CFS plánovač)
–ipbase	síť ze které se přidělují IP adresy
–mac	přiřadí deterministické MAC adresy (jinak jsou náhodné)
–switch ovsbr	použije určený typ switchů (ovsbr je Open vSwitch ve standalone módu)
–topo tree,depth=4	vybuduje stromovou topologii s hloubkou 4
–verbosity debug	nastaví podrobnost logování na určenou úroveň (debug)
–xterms	po spuštění spustí Xterm pro každý host

Tabulka 2: Příkazy v prostředí CLI Mininetu

dpctl dump-flows	vypíše všechny flow pravidla
dump	vypíše všechny prvky topologie včetně základních informací o nich
exit	ukončí Mininet
help	nápověda
h1 ip a	spustí příkaz Shellu (ip a) pod určeným hostem (h1) nebo switchem
h1 ping h2	provede ping mezi určenými hosty (z h1 na h2)
h1 wireshark-gtk &	spustí Wireshark pod určeným hostem (h1) bez blokování CLI
intfs	vypíše seznam rozhraní každého prvku
iperf h1 h2	otestuje propustnost mezi určenými hosty (h1 a h2)
links	zobrazí všechna spojení mezi prvky
link s1 h1 down	přepne link mezi prvky (s1 a h1) do požadovaného stavu (down)
net	vypíše seznam linků ke každému prvku
nodes	zobrazí všechny prvky
pingall	provede ping mezi všemi hosty
sh ovs-ofctl add-flow s1 flow	přidá zadané pravidlo (flow) na určený switch (s1)
sh ovs-ofctl dump-flows s1	vypíše všechna flow pravidla z určeného switche (s1)
sh pwd	spustí příkaz Shellu (pwd)
xterm h1 h2 h3	spustí Xterm pod určenými hosty (h1, h2 a h3)

4 MiniEdit

MiniEdit je skript, který je napsán v jazyce Python s využitím toolkitu Tk pro tvorbu GUI, původně byl vytvořen Bobem Lantzem a následně vylepšen Gregorym Geem, ale i dalšími přispěvateli. Jeho účelem je umožnit vytvoření topologie a případně i její spuštění v grafického prostředí namísto psaní příkazů do skriptu anebo na příkazovou řádku. Jsou k dispozici hosty, ze světa SDN switche a kontrolery, ze světa tradičních sítí pak klasické switche a routery (v MiniEditu mají v názvech prefix legacy, nicméne je není možné nijak konfigurovat).

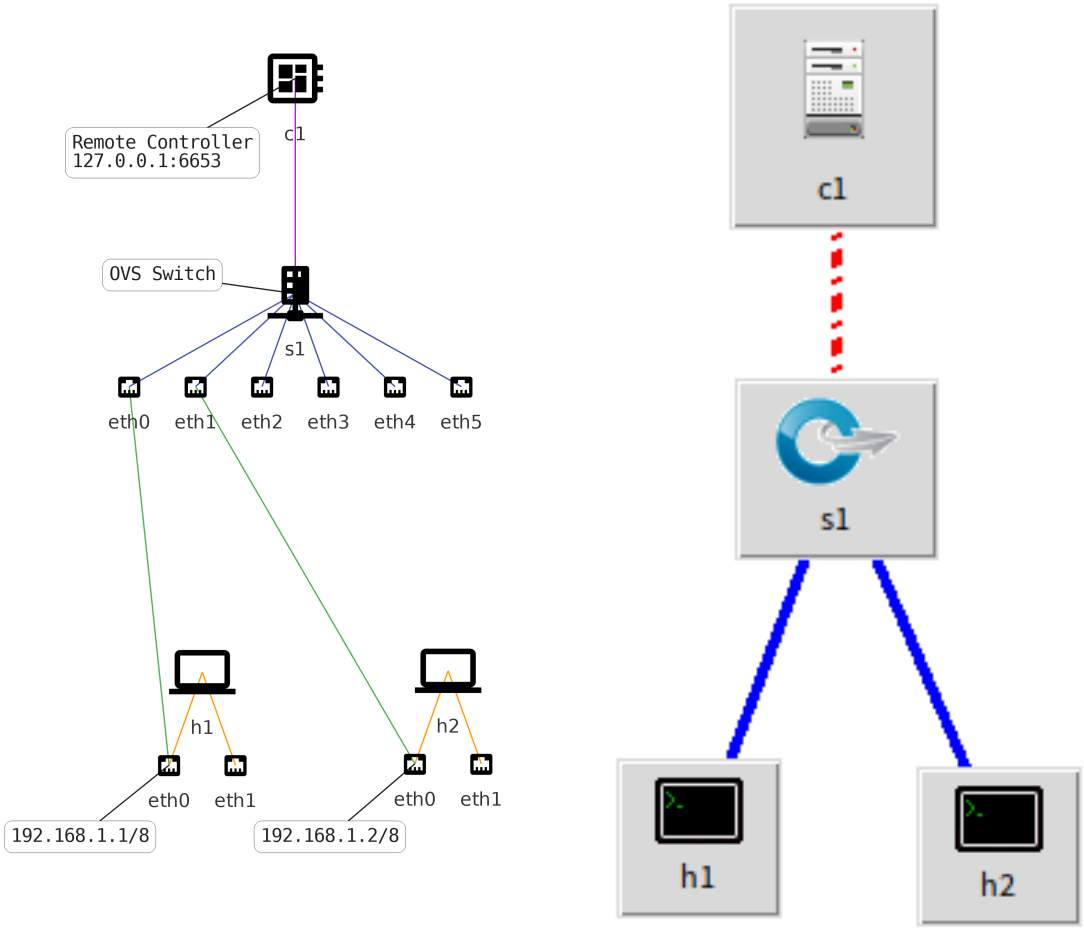


Obrázek 1: MiniEdit

Umožňuje graficky rozmístit tyto prvky a pospojovat je pomocí logických spojů (asociací) v případě propojení kontroleru se switchem nebo pomocí linků při propojování hostů a switchů (případně legacy switchů a routerů). Porty se pro spojené prvky vytváří automaticky mimo jakoukoli kontrolu uživatele (nejsou v topologii ani nijak zobrazeny). Jednotlivé položky topologie lze samozřejmě mazat, upravovat, nastavovat jejich vlastnosti a přesouvat je. Výsledný projekt je možné exportovat jako skript a také uložit do souboru pro pozdější import a další práci anebo sdílení s jinými lidmi. MiniEdit nicméně nemá příliš přívětivé GUI a není ani příliš spolehlivý, kromě exportu do JSONu anebo Pythonu a importu z JSONu již neposkytuje žádné další funkce navíc. [9]

4.1 Porovnání s Mininet editorem

MiniEdit byl inspirací pro Mininet editor, motivací pro vytvoření alternativy k němu byla zejména jeho nestabilita, neintuitivnost, uživatelská nepřívětivost a spousta dalších nedostatků. MiniEdit sice poskytuje všechnu základní funkcionalitu a je možné v něm vytvořit topologii, nastavit položky v ní, vyexportovat ji a následně spustit, uživatelský komfort je nicméně mizivý. Na obrázcích 1 a 2 si lze všimnout absence portů a z toho plynoucí nejistoty, který host je připojen ke kterému portu switche s1. Také není možné vytvořit popisky s dodatečnými informacemi jako jsou IP adresy anebo



(a) Mininet editor **(b)** MiniEdit

Obrázek 2: Porovnání stejné topologie v Mininet editoru a MiniEditu.

typy switchů a kontrolerů, ale i dalšími informacemi. Dále je také možné si všimnout nízké kvality obrázku topologie z MiniEditu, ta je zapříčiněna absencí exportu obrázků a funkce zoom.

Klady MiniEditu

- Dokáže spustit emulaci jedním kliknutím (nespolehlivé, celý MiniEdit musí běžet pod uživatelem root).
- Obsahuje legacy router (v Mininet editoru nutno vytvořit jako host s příkazem po spuštění).
- Obsahuje legacy switch (v Mininet editoru nutno vytvořit jako switch a nastavit patřičné chování).
- Součástí Mininetu (miniedit.py ve složce examples).

Zápory MiniEditu

- Je značně nestabilní.
- Legacy router ani switch se nedá konfigurovat.
- Má velmi neintuitivní ovládání.
- Nedokáže importovat skripty.
- Nedokáže vyexportovat adresní plán topologie.
- Nedokáže vyrenderovat obrázek topologie (nanejvýš snímek obrazovky s částí, která se vejde do okna).
- Nefunguje na mobilních telefonech, tabletech atd.
- Nemá názvy linků ani asociací.
- Nemá popisky.
- Nemá výběr více položek.
- Nepodporuje IPv6.
- Nepodporuje posun po plátně za hranice nejzazšího prvku.
- Nepodporuje zoom.
- Nepoužívá responzivní design.
- Neumožňuje přepojení existujících linků.
- Neumožňuje určit mezi kterými porty vede link.
- Nevyexportovaný projekt je po zavření ztracen.
- Nutné stáhnout a nainstalovat.
- Při pádu je veškerá práce ztracena (Mininet editor práci průběžně ukládá).
- Umožňuje zadat pouze jeden řádek příkazů po spuštění hostu nebo switche.
- Umožňuje zadat pouze jednu IP adresu.
- Veškerý kód je v jediném souboru (výhoda při používání, nevýhoda při vývoji).
- Vyžaduje instalaci závislostí (mimo jiného Python a Tk).
- Zadání nevalidních hodnot končí smazáním hodnoty bez zpětné vazby v GUI.

5 Mininet editor

Stěžejní částí této diplomové práce je Mininet editor, aplikace, která umožňuje tvorbu, export a import topologie pro Mininet v grafickém rozhraní. Mininet editor byl inspirován již existujícím MiniEditem (více v kapitole 4), jehož skripty je schopen importovat (více v kapitole 5.4.5).

Pro vývoj Mininet editoru bylo použito metod agilního vývoje. Samotná aplikace nejprve vznikla jako ukázkový prototyp se zlomkem funkcionality, který byl následně konzultován s vedoucím práce. Připomínky a chybějící funkcionalita byly po každé konzultaci zapracovávány do aplikace v rámci obvykle týdenních sprintů.

5.1 Požadavky

Tato kapitola obsahuje seznam požadavků, které jsou kladeny na Mininet editor.

Funkční požadavky

1. Aplikace umožní definici vlastních Mininet skriptů spuštěných po startu nebo před vypnutím prvku a Mininetu.
2. Aplikace umožní export adresního plánu ve formátu PDF.
3. Aplikace umožní export obrázku topologie ve formátu PNG.
4. Aplikace umožní export pro emulaci topologie v Mininetu do skriptu v jazyce Python.
5. Aplikace umožní export vytvořené topologie pro pozdější import ve formátu JSON.
6. Aplikace umožní import exportované topologie.
7. Aplikace umožní import skriptu exportovaného z MiniEditu.
8. Aplikace umožní import skriptu exportovaného z této aplikace.
9. Aplikace umožní import ukázkové topologie z integrovaného seznamu.
10. Aplikace umožní mazání položek z topologie.
11. Aplikace umožní propojení hostů a switchů přes specifické porty.
12. Aplikace umožní přidávání libovolných popisků do topologie.
13. Aplikace umožní vrácení provedených změn v topologii (funkce zpět/undo).

14. Aplikace umožní vytvoření topologie v GUI.
15. Aplikace umožní základní konfiguraci Mininetu.
16. Aplikace umožní základní konfiguraci položek v topologii.

Nefunkční požadavky

1. Aplikace bude dodržovat responzivní design.
2. Aplikace bude PWA.
3. Aplikace bude SPA.
4. Aplikace bude zveřejněna s otevřeným zdrojovým kódem.
5. Aplikace poběží na statickém web serveru.
6. Aplikace si bude uchovávat data o topologii v perzistentní paměti.

5.2 Struktura projektu

Cílem této kapitoly je popsání struktury projektu z hlediska rozmístění zdrojových kódů a dalších souborů v adresářové struktuře projektu. Tyto soubory jsou rozčleněny do složek, každá ze složek pak odpovídá určité funkcionalitě.

./ V kořenové složce se nachází pouze různé konfigurační soubory, licence projektu a soubor s krátkým popisem projektu pro zobrazení na hlavní stránce repozitáře na Githubu.

public Ikony stránky v různých formátech pro různá zařízení, prohlížeče a operační systémy, webmanifest a šablona stránky.

src Kořenová složka zdrojových kódů. Nachází se zde pouze kód sloužící k inicializaci frameworku, ostatní funkcionalita je umístěna v podsložkách.

src/assets Grafické prvky aplikace, jmenovitě ikona aplikace a obrázky prvků topologie (kontroler, switch, host, port). Ostatní grafické prvky jsou součástí použitých knihoven, ze kterých pochází i prvky topologie, nicméně jejich renderování na plátno vyžaduje, aby tyto obrázky byly k dispozici ve formě souborů.

src/builder Sestavení Python skriptu a adresního plánu z formátu JSON.

src/components Komponenty GUI.

src/components/edit Subkomponenty a moduly pro komponentu Edit.vue. Obsahuje specifické komponenty pro editaci jednotlivých typů položek (kontrolery, linky, atp.) topologie.

src/components/export Subkomponenty a moduly pro komponentu Export.vue. Obsahuje komponenty pro import, export a zobrazení logu těchto operací, ty jsou sestaveny do jedné stránky komponentou Export.vue.

src/components/vis Subkomponenty a moduly pro komponentu Vis.vue. Tato komponenta je z důvodu své komplexnosti rozdělena na několik podčástí.

src/examples Ukázkové projekty a prázdný projekt.

src/exporter Převod vnitřní reprezentace projektu do formátu JSON. Použit mimo jiného jako rozhraní pro výměnu dat mezi moduly exportu, importu, persistence a zbytkem aplikace.

src/importScript Tvorba projektu ve formátu JSON na základě skriptu v Pythonu.

src/router Správa URL adresy.

src/store Centrální úložiště dat. Zajišťuje sdílení reaktivních dat napříč aplikací, zaznamenávání změn prováděních v topologii a držení jejich historie, zpět/vpřed funkcionalitu a persistenci dat projektu.

src/validation Ověřovaní vstupních dat zadaných uživatelem.

tests/e2e Integrační testy (Cypress).

tests/unit Jednotkové testy (Mocha/Chai).

5.3 Použité technologie

V této kapitole jsou popsány důležité technologie, které byly použity pro vývoj Mininet editoru. U každé technologie je uveden její popis a k čemu byla v rámci projektu použita. Větší prostor je věnován zásadním technologiím, které zajišťují klíčovou funkcionalitu anebo jsou použity napříč celou aplikací. Menší množství prostoru je pak věnováno méně důležitým technologiím a některé, které nejsou použity pro zásadní funkcionalitu, zde nejsou zmíněny vůbec.

ECMAScript Tento jazyk vznikl standardizací jazyků JavaScript (NetScape), JScript (Microsoft) a dalších variant. I když je pro něj vetšinou používán název JavaScript, který byl prvním zveřejněným názvem tohoto jazyku, tento termín je registrovanou ochranou známko společnosti Oracle [10] a proto se neobjevuje v mnoha oficiálních textech. Vývoj v dnešní době probíhá otevřeně na Githubu na základě nahlášených chyb, pull requestů a experimentování s nestandardními rozšířeními v různých implementacích standardu.

Moderní ECMAScript (ES2018) je vysoce expresivní jazyk s velkým množstvím jazykových konstrukcí jako jsou async funkce, try-catch-finally, promise, destructuring, spread a další. Zpětná kompatibilita se staršími implementacemi se často dá zajistit pomocí polyfilů a transpilace pro starší verze standardu, které mají lepší podporu v běžně používaných enginech (ve starších ES3, v novějších ES5). Za dobu své existence našel ECMAScript uplatnění nejen v segmentu webových stránek, pro který byl původně vytvořen, ale i v mnoha dalších oblastech. [11]

Téměř celá práce je vypracována v ECMAScriptu s výjimkou pár řádků Pythonu v exportovaných skriptech.

Vue.js Progresivní framework pro tvorbu uživatelského rozhraní se zaměřením na single page aplikace. Vue samo o sobě zajišťuje pouze rozdělení aplikace na komponenty, komunikaci mezi komponentami, reaktivitu dat a jejich navázání na HTML template. Veškerou další funkcionalitu zajišťují jiné projekty z ekosystému, které se v případě potřeby integrují s Vue. [12]

Celá GUI část Mininet editoru je postavena nad Vue frameworkem.

Vuex Centrální úložiště runtime dat pro Vue. Umožňuje udržovat a sdílet data napříč celou aplikací, zajišťuje reaktivitu dat a při vývoji zaznamenává každou změnu a umožňuje tak cestovat v čase a sledovat při tom vývoj změn. [13]

Veškerá sdílená data, jako je například topologie nebo nastavení Mininetu, jsou v Mininet editoru uložena ve Vuex úložišti.

Vuetify.js Knihovna znovupoužitelných komponent pro Vue framework v Material Design stylu. Komponenty jsou responzivní a optimalizované jak pro desktopová a mobilní zařízení, tak i pro různé prohlížeče na každé platformě a v neposlední řadě i pro server side rendering. [14]

Vuetify komponenty jsou použity napříč aplikací od základního rozvržení až po jednotlivá tlačítka a vstupy.

vis.js Dynamická knihovna pro grafické zobrazování velkého množství dat ve webovém prohlížeči za pomocí plátna (canvas). Data lze zobrazit ve formě 2D a 3D grafů, na časové ose nebo jako síťovou topologii. Umožňuje také interaktivní zásahy ze strany uživatele a libovolné úpravy dat v reálném čase i s animacemi přechodů mezi starými a novými daty. [15]

Plátno s mapou sítě je renderováno pomocí knihovny vis.js.

Material Design Icons Celkem bohatá open source kolekce ikon, která je spravována komunitou. [16]

Všechny ikony v Mininet editoru, s výjimkou loga, pocházejí z tohoto setu.

ANTLR Mocný nástroj pro čtení, zpracovávání, vykonávání nebo překlad strukturovaného textu nebo binárních souborů. [17]

V práci je použit pro vygenerování abstraktního syntaktického stromu z Python skriptu pro usnadnění jeho zpracování při importu.

jsPDF Nástroj pro generování PDF dokumentů přímo ve webovém prohlížeči anebo na platformě Node. [18]

Export adresního plánu využívá jsPDF pro tvorbu a uložení PDF dokumentu.

Webpack Modulární nástroj pro sestavování aplikací. Umožňuje zabalit jednotlivé moduly do jednoho nebo více souborů pro rychlejší načítání přes Internet a to i při kombinaci různých typů modulů. Dále umožňuje předzpracovávat vstupní data, například transpilovat TypeScript do JavaScriptu, vkládat obrázky přímo do HTML/CSS ve formátu Base64, atd. [19]

Webpack je použit pro sestavení aplikace (jak při vývoji tak i pro zveřejnění produkční verze).

Babel Nástroj, pro transpilaci moderního JavaScriptu (ES2018 v době tvorby aplikace) do verze ES5, která je běžně podporována. [20]

Celý Mininet editor je pomocí Babelu transpilována do ES5, aby byla zajištěna kompatibilita s velkým množstvím prohlížečů i když jsou ve zdrojových kódech využívány konstrukce z verze ES2018.

Mocha Testovací framework pro JavaScript, který může běžet jak v prostředí webového prohlížeče tak i v prostředí Node. Mezi jeho funkce patří podpora pro asynchronní testy, transpilery, různé assertion knihovny, atd. [21]

Unit testy jsou napsány s pomocí tohoto frameworku.

Chai Assertion knihovna, kterou lze použít s frameworkem Mocha. Poskytuje should, expect a assert assertion styly. [22]

Expect z této knihovny je použit v unit testech.

Cypress Framework pro integrační testování frond endu. Mezi jeho funkce patří cestování v čase (zobrazení stránky tak, jak vypadala v průběhu testu), automatické čekání na události v testované aplikaci, podpora pro práci se sítí v testech, snímky obrazovky a videa při selhání testu a další. [23]

Cypress je použit pro integrační testování sestavené aplikace.

5.4 Implementace

Tato kapitola popisuje implementaci Mininet editoru včetně fragmentů kódu stěžejních částí aplikace a objasnění jejich fungování. Cílem je představení implementace pro usnadnění orientace v kódu a případný další vývoj aplikace.

5.4.1 Plátno

Stěžejní částí Mininet editoru je plátno (canvas) na kterém je vytvářena topologie. Renderování je z vetší části zajištěno pomocí knihovny Vis, ale některé funkce, jako je výběr pravým tlačítkem myši, bylo nutné doprogramovat. Knihovna Vis pro takovéto případy umožňuje zaregistrovat listenery (jeden z nich je možné vidět ve fragmentu kódu 1), které pak mohou dodatečně vykreslovat na plátno aniž by jejich činnost kolidovala s vykreslováním knihovny.

```
1  _afterDrawingListener (ctx) {
2    if (this._drag) {
3      const { startX, startY, endX, endY } = this._rectCanvas
4
5      ctx.lineWidth = 4
6      ctx.strokeStyle = this._colors.border
7      ctx.strokeRect(startX, startY, endX - startX, endY - startY)
8      ctx.fillStyle = this._colors.background
9      ctx.fillRect(startX, startY, endX - startX, endY - startY)
10   }
11 }
```

Fragment kódu 1: Listener pro vykreslování hranic výběru

Implementace plátna je rozdělena na interaktivní a neinteraktivní část. Neinteraktivní část slouží pouze k renderování topologie z úložiště. Interaktivní část zajišťuje interakce s uživatelem (přidávání položek, mazání, posuny atp.), které ukládá do úložiště, a využívá při tom neinteraktivní část pro renderování. Díky tomu, že se změny nejdříve promítají do úložiště, není možné, aby došlo k jakémukoli rozdílu mezi stavem úložiště a zobrazenou topologií. Uživatel tedy vždy vidí přesně to, co je uloženo, a v průběhu vývoje jsou okamžitě zjevné jakékoli chyby v ukládání změn anebo ren-

derování topologie. Také to umožňuje případné rozšíření aplikace například modulem pro napojení na živou emulaci v Mininetu, kterému by pak stačilo jen sledovat a reagovat na změny v úložišti, případně je provádět bez zásahů do kódu zbytku aplikace. Takovýto modul je nicméně nad rámec této diplomové práce.

5.4.2 Úložiště

Veškerá data projektu jsou uložena v úložišti Vuex, to zajišťuje jejich reaktivitu, možnost sledování změn a, za pomocí pluginu, i persistenci. Data projektu jsou persistentně uložena ve formátu JSON (ukázka ve fragmentu kódu 5), ten obsahuje mimo jiného i verzi a je nezávislý na používané reprezentaci v rámci operační paměti. Perzistentní data je tedy v případě potřeby možné při obnovení i převést na nový formát bez ztráty rozpracovaného projektu anebo porušení jeho integrity.

V úložišti je kromě samotných dat projektu (topologie a nastavení) uložena ještě historie změn pro funkce zpět a vpřed (undo/redo). Při každé změně v topologii se uloží rozdíl pro obnovení stavu před danou změnou anebo pro znovuprovedení dané změny, přičemž množství uchovávaných změn je omezeno na 200. Historie je, stejně jako projekt, perzistentní, tedy zůstává zachována i při zavření a opětovném načtení stránky, není ale součástí exportu.

Pro čtení je možné k datům v úložišti přistupovat přímo. Alternativně lze nadefinovat gettery, jejichž hodnoty jsou vypočteny při prvním přístupu a pak uchovávány v paměti a přepočteny jsou jen v případě, že dojde ke změně dat ze kterých byly původně vypočteny. Sledování změn pomocí vestavěné reaktivity frameworku Vue funguje jak při přímém používání hodnot tak i při používání getterů. Obě výše zmíněné metody jsou použity pro zobrazování sdílených dat napříč celou aplikací. Třetí možností jak přistupovat k datům je naslouchání změnám, tato metoda je využita pro aktualizaci plátna při změně položek topologie.

Změny dat se smí provádět pouze z mutací, to zajišťuje, že data mohou být měněna jen přesně definovanými způsoby. Toto omezení také umožňuje při vývoji zaznamenávat stav před každou změnou a cestovat časem pro snazší hledání chyb v aplikaci. Mutace nicméně musí být synchronní funkce, nelze v nich tedy vykonávat asynchronní kód, ten by totiž způsobil, že by mutace nebyly atomické. To by, vzhledem k tomu, že je úložiště přístupné ze všech částí aplikace, značně ztížilo zajištění konzistentního

stavu úložiště. Pokud je potřeba provádět asynchronní operace v úložišti, je nutné použít akce, které mohou být asynchronní, ale musí používat mutace pro změnu stavu. Lze tedy například operaci rozdělit do více mutací anebo v akci nejprve shromáždit data z asynchronních zdrojů a až poté provést jednu synchronní mutaci.

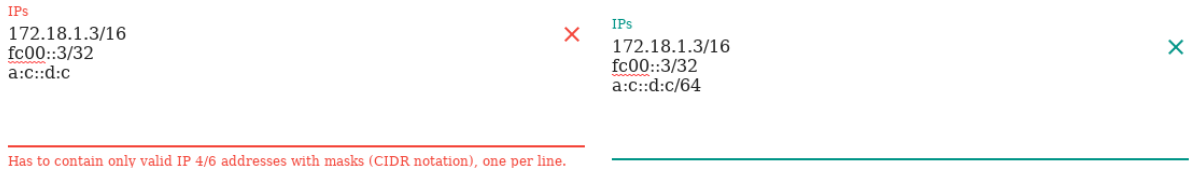
```
removeItems ({ commit, state }, ids) {
  commit('pushChange', ids.map(id => ({
    before: JSON.stringify(state.data.items[id] || null),
    after: JSON.stringify(null)
  })))

  commit('applyChange', {
    remove: ids
  })
},
```

Fragment kódu 2: Uložení dat pro funkci zpět a jejich odstranění z topologie

Ve fragmentu kódu 2 je vyobrazena akce, která slouží k mazání položek z topologie. Tato akce je synchronní, slouží pouze k seskupení dvou mutací, které se používají pomocí funkce `commit`. V první části provede uložení stavu pomocí samostatné mutace, tento stav obsahuje dva seznamy. Jednak stav před smazáním, tedy seznam položek, které byly smazány, ty se pak dají případně obnovit pomocí funkce zpět. Dále také stav po smazání, tedy seznam s odpovídající hodnotou null (někdy nazývanou jako záměrně neexistující objekt v opozici k hodnotě undefined) pro každou smazanou položku. Tyto hodnoty jsou serializovány do formátu JSON, tato metoda byla zvolena proto, že umožňuje jednoduše a rychle vytvořit hlubokou kopii dat, jako další důvod se dá uvést jeho široká podpora i v obstarožních prohlížečích. Vysoká rychlost je zde dána jednak tím, že JSON je podmnožinou JavaScriptu, a také tím, že je implementován nativně v prohlížečích. V druhé části jsou pak provedeny změny v úložišti, tedy samotné smazání položek z topologie. Tato operace, protože mění stav úložiště, je opět definována ve své vlastní mutaci.

5.4.3 Validace

Všechny hodnoty jsou validovány již při zadávání do aplikace, jak je možné vidět na obrázku 3, zpětná vazba je poskytována v reálném čase. Název a další prvky pole se přebarví na zeleno, pokud jsou hodnoty validní, nebo na červeno a pod polem se objeví popis chyby, pokud hodnoty validní nejsou. GUI také nedovolí ukládání nevalidních hodnot, v tomto případě nelze do portu uložit IPv6 adresu bez masky.



(a) Nevalidní **(b)** Validní

Obrázek 3: Validace zadaných hodnot

Pro validaci byla použita jednak vestavěná pravidla, dále bylo potřeba nadefinovat speciální funkce, protože vestavěná pravidla neobsahují mimo jiného IP adresy, porty a další pravidla pro specifické formáty používané v parametrech metod Mininetu. Vstupem všech pravidel jsou textové řetězce zadané uživatelem (ve fragmentu kódu 3 jako parametr v). Výstupem je pak hodnota true pro vyhovující řetězce nebo false pro nevyhovující. Zobrazení chybové hlášky pod textovým polem v GUI je vestavěnou funkcí frameworku Vuetify.

```
1  const testMask6 = v => /^\d+$/.test(v) && v >= 0 && v <= 128
2
3  const testIP6 = v => {
4    if (!/^[a-fA-F0-9:]+$/.test(v)) {
5      return false
6    }
7
8    const parts = v.split(':')
9    if (!parts.every(p => p.length <= 4)) {
10     return false
11   }
12
13   const every = parts.length
14   const empty = parts.filter(p => p.length === 0).length
```

```
15    return (empty === 0 && every === 8) ||
16      (empty === 1 && every <= 8)
17  }
18
19  const testIP6WithMask = v => {
20    const [ip, mask, tail] = v.split('/')
21    return tail === undefined &&
22      ip != null && testIP6(ip) &&
23      mask != null && testMask6(mask)
24  }
```

Fragment kódu 3: Ukázka pravidel validace

Dále se také validují propojení jednotlivých prvků v topologii. Linky mohou spojovat pouze porty, protože odpovídají fyzickým spojům v simulované síti a tím pádem jiné varianty ani nedávají smysl. Asociace spojují zařízení pouze logicky, byť například spojení switche s kontrolerem může odpovídat fyzickému kabelu ve skutečné síti. U každé asociace je kontrolováno, jestli dává smysl. Asociace kontroler na switch znamená, že kontroler ovládá switch. Asociace switch případně host na port znamená, že daný prvek má daný port. Popisek (ve fragmentu kódu 4 jako dummy) je možné připojit na cokoli. Jiné asociace jako například switch na host nebo jeden kontroler na jiný kontroler smysl nedávají a proto je nelze vytvořit.

```
1  const edgeTests = {
2    'link': (src, dst) => src === 'port' && dst === 'port',
3    'association': (src, dst) => (
4      (src === 'controller' && dst === 'switch') ||
5      (src === 'switch' && dst === 'port') ||
6      (src === 'host' && dst === 'port') ||
7      (src === 'dummy')
8    )
9  }
```

Fragment kódu 4: Ukázka validace propojení v topologii

5.4.4 Export a import projektu

Export ve formátu JSON slouží pro uložení kompletního projektu včetně všech metadat, který může být znovu importován bez ztráty jakýchkoli dat.

```
{
    "startScript": "pingall\n",
    "projectName": "Mininet network",
    "ipBase": "192.168.1.0/8",
    "listenPortBase": 6634,
    "autoSetMAC": true,
    "spawnTerminals": true,
    "inNamespace": false,
    "autoStaticARP": false,
    "logLevel": "debug",
    "version": 0,
    "items": [{
        "id": "adc7b0a0-51e5-4678-a4bc-458ac91baed5",
        "type": "switch",
        "hostname": "s1",
        "switchType": "OVSBridge",
        "x": -30,
        "y": -161
    }, {
        "id": "5f9035bb-423b-4399-9d5a-92cf0d2e2d25",
        "hostname": "eth0",
        "type": "port",
        "x": -155,
        "y": -91
    }, {
        "id": "1fc7e520-980e-44b4-80e4-cf54dbb759b6",
        "type": "association",
        "from": "adc7b0a0-51e5-4678-a4bc-458ac91baed5",
        "to": "5f9035bb-423b-4399-9d5a-92cf0d2e2d25"
    }, {
```

Fragment kódu 5: Ukázka exportu ve formátu JSON

Každý soubor povinně obsahuje verzi, ta je zde přítomna kvůli případným změnám ve formátu, které by porušily zpětnou kompatibilitu, a seznam položek v topologii, ten může být prázdný. Dále obsahuje konfiguraci, ve formátu klíč/hodnota, která bude při spuštění předána Mininetu. Při její absenci budou dané hodnoty v rámci aplikace zobrazené jako prázdné, při spuštění v Mininetu se použijí výchozí hodnoty.

Formát jednotlivých položek se liší podle jejich typu, nicméně každá má povinně id a povinně nebo volitelně (podle typu) hostname, který má další omezení u položek, kde se používá jako identifikátor ve skriptu. Aplikace jako id generuje UUID, ale nejsou kladeny žádná zvláštní omezení s výjimkou jedinečnosti, jakýkoli textový řetězec je akceptovatelný. Jednotlivé položky pak mají další hodnoty, v případě hran jsou to povinně id položek, které spojují. Linky mají navíc ještě volitelné údaje omezení provozu. Hosty, switche a kontrolery obsahují volitelné údaje pro Mininet, například IP adresu a port u kontrolerů, typ u switchů nebo výchozí cestu u hostů.

Tento formát také slouží jako rozhraní pro export a import skriptů v Pythonu a export adresního plánu. Díky čemuž tyto funkcionality nemají těsnou vazbu na formu dat použitou v úložišti.

5.4.5 Export skriptu

Export ve formě Python skriptu slouží primárně pro spouštění v Mininetu. Vyexportovaný skript obsahuje vše od spuštění Mininetu, přes sestavení topologie, nastavení adres až po otevření CLI pro další práci. Jediné, co je potřeba spustit ručně je Open vSwitch Daemon a případné kontrolery použité v topologii.

Pro každý host, switch a kontroler je ve skriptu vytvořena proměnná, jako její název je použit vždy hostname příslušného prvku. Z toho samozřejmě vyplývají určitá omezení, například že hostname nesmí obsahovat mezery nebo že se skript nevyexportuje pro topologii, ve které nemá každý prvek unikátní hostname.

```
1 # Add nodes {{{
2
3 mininet.log.info('\n*** Add nodes\n')
4
5 c1 = net.addController(
6         'c1',
```

```
7           controller=mininet.node.RemoteController,
8           ip='127.0.0.1',
9           port=6653
10  )
11  h1 = net.addHost('h1', ip=None)
12  h2 = net.addHost('h2', ip=None)
13  s1 = net.addSwitch('s1', cls=mininet.node.OVSSwitch)
14
15  # }}}
16  # Add links {{{
17
18  mininet.log.info('\n*** Add links\n')
19
20  net.addLink(h1, s1, intfName1='h1-eth0', intfName2='s1-eth0')
21  net.addLink(h2, s1, intfName1='h2-eth0', intfName2='s1-eth1')
22
23  # }}}
```

Fragment kódu 6: Ukázka exportu ve formátu Python

Vyexportovaný skript je rozdělen na sekce (ty se dají i zavřít, pokud to podporuje použitý editor), ve fragmentu kódu 6 jsou vidět dvě. Prvním je přidání prvků, v tomto případě jednoho kontroleru, dvou hostů a jednoho switche. Lze si všimnout, že u hostů nejsou nastaveny IP adresy. Je to z toho důvodu, že Mininet neumožňuje nastavení více než jedné IP adresy a ta ještě musí být IPv4, proto jsou adresy nastaveny později pomocí příkazu `ip`. Switchi je nastaven typ (OVS Switch) a u kontroleru ještě navíc IP adresa (127.0.0.1) a port (6653). Další je ve fragmentu přidání linků mezi hosty. U každého linku je definováno které prvky spojuje a názvy jejich rozhraní. Názvy rozhraní jsou vytvořeny podle konvencí Mininetu, první částí je hostname příslušného prvku a druhou částí je název portu, tyto části jsou spojeny pomocí pomlčky (technicky vzato se jedná o znak `U+002D`, tedy o spojovník). Výslovné uvedení názvu každého rozhraní umožňuje spolehlivě spojit IP adresy s příslušnými porty tak, jak je to uvedené v exportované topologii.

Z programového hlediska je skript nejprve vytvořen ve formě několika polí, kde každé pole odpovídá určité sekci skriptu. Tyto sekce se až na konec spojí do jednoho textového řetězce, který se následně vyexportuje ve formě souboru. Takovéto uspořádání

usnadňuje generování skriptu, protože není nutné generovat skript řádek po řádku tak, jak se bude vykonávat.

```
const args = pyArgs([
  [host.hostname, String],
  ['None', null, 'ip'],
  [host.defaultRoute != null, `via ${host.defaultRoute}`,
    String, 'defaultRoute'],
  [
    [host.cpuScheduler, host.cpuCores, host.cpuLimit].some(v => v != null),
    'mininet.node.CPULimitedHost', null, 'cls'
  ]
])
this._code.nodes.push(`${host.hostname} = net.addHost(${args.join(', ')})`)

if (host.cpuScheduler != null || host.cpuLimit != null) {
  const args = pyArgs([
    [host.cpuScheduler != null, host.cpuScheduler, String, 'sched'],
    [host.cpuLimit != null, host.cpuLimit, Number, 'f']
  ])
  this._code.nodeLimits.push(
    `${host.hostname}.setCPUFrac(${args.join(', ')})`
  )
}
if (host.cpuCores != null) {
  const args = pyArgs([
    [host.cpuCores != null, host.cpuCores.join(','), String, 'cores']
  ])
  this._code.nodeLimits.push(
    `${host.hostname}.setCPUs(${args.join(', ')})`
  )
}

this._addNodeScripts(host.hostname, host.startScript, host.stopScript)
```

Fragment kódu 7: Export hostu do skriptu

Ve fragmentu kódu 7 se nachází tělo funkce, která slouží ke zpracování hostů při exportu skriptu. Kód využívá pomocnou funkci pyArgs, která vytváří pole argumentů

ve formátu používaném Pythonem (například argument určující název rozhraní prvního prvku při vytváření linku `intfName1='h2-eth0'` nebo třída použitá pro inicializaci switche `cls=mininet.node.OVSSwitch` ve fragmentu kódu 6), ty pak stačí spojit do textového řetězce odděleného čárkami a vložit do kulatých závorek při volání funkce. V kódu si lze všimnout, že host může mít několik různých operací v exportovaném souboru a to vytvoření s parametry, nastavení plánovače, nastavení CPU a vykonání skriptů. Důležité je zachování pořadí jednotlivých operací, nastavení plánovače a CPU totiž nelze provést před vytvořením hostu, spouštěcí skript nemůže být vykonán před sestavením topologie a vypínací skript se musí vykonat až poté, co uživatel ukončí CLI sezení. Zde je krásně vidět výhoda zvoleného řešení, protože host se dá zpracovat v jednom průchodu a jednotlivé operace budou ve vhodném pořadí, protože se nachází v různých sekcích jejichž pořadí je dané a bude dodrženo při spojení do jednoho textového řetězce. Dále je možné si všimnout, že IP adresa je staticky nastavena na hodnotu `None`, je to z toho důvodu, že IP adresy se vážou k jednotlivým portům, nikoli přímo k hostu.

Po projití všech položek a nastavení z projektu dojde ke spojení všech řádků do jednoho řetězce a to s tím, že bude dodrženo definované pořadí jednotlivých sekcí. Při spojení jsou jednotlivé sekce pro lepší přehlednost opatřeny komentáři a volitelně i logovacími zprávami. Pokud je určitá sekce prázdná, tak se z výsledného skriptu vypouští. K sekcím je ještě připojena hlavička a následně je vše spojeno do jednoho textového řetězce, který jíž obsahuje funkční skript v Pythonu.

```
1  toString () {
2    const code = []
3    metadata.forEach(({ attr, name, silent }) => {
4      const arr = this[attr]
5        .map(v => v.apply ? v.apply() : v)
6
7      if (arr.length) {
8        code.push(
9          `# ${name} {{{`,
10         '',
11         ...(silent ? [] : [
12           `mininet.log.info('\\n*** ${name}\\n')`,
13           ''
14         ]),
```

```
15          ...arr,
16          '',
17          '# }}}'
18        )
19      }
20    })
21
22    return [
23      '#!/usr/bin/env python2',
24      '# -*- coding: utf-8 -*-',
25      '',
26      ...code,
27      '',
28      '# vim:fdm=marker',
29      ''
30    ].join('\n')
31  }
```

Fragment kódu 8: Spojení polí do stringu při exportu do skriptu

5.4.6 Import skriptu

Exportovaný skript je, kromě použití pro spuštění emulace, možné i importovat zpět do Mininet editoru, v takovém případě ale není možné obnovit všechna data původního projektu. Zejména se jedná o popisky a polohy jednotlivých položek, součástí skriptu ale nejsou ani nepřipojené porty. Import je testovaný pouze pro skripty exportované Mininet editorem a MiniEditem, kde funguje spolehlivě. Skripty, lze importovat, i v případě, že nebyly vytvořeny jednou z těchto aplikací anebo že byly následně upravovány, je ale nezbytné vzít v potaz, že takový skript nemusí být importován korektně. Importovaný kód se totiž nevykonává a proto různé cykly, proměnné a další konstrukce jazyka nebudou korektně vyhodnoceny. Také nelze importovat kód z více soubory, byť je možné vše spojit do jednoho a ten následně importovat, neexistuje ale záruka, že import proběhne korektně.

Při vývoji bylo uvažováno více potenciálních metod importu skriptů. Tou nejjednodušší je analýza souboru pomocí regulérních výrazů, ta byla nicméně zavržena ještě před napsáním prvních řádek kódu. Její fatální slabinou je skutečnost, že nerozumí

syntaxi analyzovaného kódu, a tím pádem je velmi nespolehlivá například v případě, že je použito jiné formátování kódu. Je také velmi obtížné takovýto kód dlouhodobě udržovat, regulérní výrazy se totiž s rostoucí komplexitou velmi rychle stávají jen obtížně srozumitelnými.

Druhou zvažovanou metodou bylo vytvoření falešného Mininetu v prostředí webového prohlížeče a následné vykonání kódu za pomoci překladu Pythonu do JavaScriptu. Při použití tohoto přístupu, by se při při volání metod Mininetu namísto emulace sítě pouze zaznamenávaly požadované operace a na jejich základě, by byl následně sestaven projekt. Nespornou výhodou je zde skutečné vykonání skriptu, které umožňuje například i generování položek topologie pomocí cyklů, výrazů a podmínek, a oproti variantě s využitím regulérních výrazů i nezávislost na formátování kódu. Na druhé straně je zde ale velké množství nevýhod, jednou z nich, byť není fatální, je i podstatně vyšší náročnost implementace takovéhoto řešení. Tato náročnost vyplývá také z toho, že je nutné vytvořit nejen výše zmíněné metody, ale také jejich návratové hodnoty, a to vše tak, aby kód ve skriptu nenarazil na chybějící metodu nebo nedostal nesmyslnou návratovou hodnotu, která by mohla vavolat výjimku. Horší je skutečnost, že při výskytu jakékoli výjimky ve vykonávaném kódu by došlo k ukončení a nenaimportování skriptu, respektive naimportování jen jeho začátku. Tento problém je ještě umocněn překladem Pythonu do JavaScriptu a z toho vyplývajícími omezeními. Některé moduly Pythonu totiž nebyly pro vykonávání v prohlížeči nikdy implementovány a jiné ani být implementovány nemohou, například čtení souborů musí v prostředí prohlížeče vždy selhat, jelikož webové stránky nemají přímý přístup k systému souborů na počítači. Z toho vyplývá, že by tato metoda velmi snadno selhávala pro skripty napsané nebo upravené uživately a to často již při importování modulů, díky čemuž by došlo k importování pouze prázdného projektu. Pro import skriptů generovaných aplikacemi Mininet editor a MiniEdit, jejichž import je hlavním důvodem existence této funkcionality, tato metoda, vzhledem k povaze jimi generovaného kódu, nenabízí žádné výhody.

Metoda, která byla zvolena pro import skriptů, využívá parsování skriptu a následnou analýzu abstraktního syntaktického stromu. Tato metoda sice nedokáže zpracovat cykly, podmínky a další konstrukce, její výhoda ale spočívá v odolnosti proti chybám, neznámým konstrukcím nebo importům ve skriptu, ty se jednoduše přeskočí. Díky

tomu dokáže importovat i skripty, které obsahují velké množství kódu přidaného uživatelem, přitom stačí, aby byl skript syntakticky validní. Z hlediska hlavního účelu importu skriptů, tedy skriptů generovaných aplikacemi Mininet editor a MiniEdit, je tato metoda nejvýhodnějším řešením. Pro její hlavní účel je totiž podstatně spolehlivější než ostatní metody, snáze udržovatelná než metoda využívající regulérní výrazy a podstatně rychlejší a jednodušší na implementaci než kompilace Pythonu do JavaScriptu a následné vykonávání skriptu a odchytávání volání metod Mininetu.

Z hlediska implementace importu skriptů je pro samotné parsování skriptu využita knihovna ANTLR, která zajišťuje vytvoření abstraktního syntaktického stromu z kódu v Pythonu. Kód je v rámci stromu následně procházen shora dolů a nalezené výrazy jsou analyzovány. Pokud jednotlivé výrazy odpovídají známým konstrukcím, tak jsou z nich sestavovány odpovídající položky topologie a vztahy mezi nimi. Z tohoto vyplývá, jak již bylo řečeno výše, že kód není vykonáván a proto ani nedochází k provádění cyklů, vyhodnocování výrazů, větvení a dalších konstrukcí jazyka. Nicméně jsou zaznamenávány proměnné, byť tak jak jsou definované v souboru, nikoli tak, jak by byly vykonané v rámci běhu skriptu, a jejich hodnoty jsou následně dosazovány na místa, kde jsou použity. Tohle ale nefunguje v případě, že proměnné obsahují hodnoty, které nejsou napsané přímo v kódu. Hodnoty proměnných, které jsou výsledkem volání funkcí, matematických výrazů atp. vyvolají chybu a nebudou zpracovány. Každé takovéto selhání je zapsáno do logu a import pak pokračuje dál, což může způsobit další selhání v následujících částech skriptu, kde jsou dané proměnné použity, a nenačtení dalších položek z něj.

```
} else if (funcName === '.addController') {
  // Controller
  const hostname = pyString(args[0] || args.name)
  const item = {
    type: 'controller',
    hostname
  }
  if (pyNotNull(args.controller)) {
    item.controllerType = args.controller.replace(/.*\./, '')
  }
  if (pyNotNull(args.ip)) {
```

```
12      item.ip = pyString(args.ip)
13    }
14    if (pyNotNull(args.port)) {
15      item.port = pyNumber(args.port)
16    }
17    if (pyNotNull(args.protocol)) {
18      item.protocol = pyString(args.protocol)
19    }
20
21    items.put(item)
```

Fragment kódu 9: Import volání metody addController ve skriptu

Ve fragmentu kódu 9 se nachází ukázka, jak probíhá zpracování volání metody, která přidává do topologie nový kontroler, její název ve skriptu je `addController`. Její název a argumenty byly zpracovány z výstupu nástroje ANTLR, který vytvořil textové reprezentace jednotlivých jednotek kódu. Vzhledem k tomu, že každá jednotka je string, nehledě na datový typ, který reprezentuje, bylo nutné vytvořit pomocné funkce k převodu a testování těchto hodnot. Ve fragmentu jsou použity funkce `pyString` a `pyNumber`, které převádí reprezentace ve formě textových řetězců na nativní datové typy JavaScriptu, a dále funkce `pyNotNull`, která kontroluje, jestli je hodnota platná. Na samotném konci fragmentu je kontroler přidán do seznamu položek. Jakákoli výjimka v tomto kódu je odchycena a nezpůsobí selhání celého importu, pouze nedojde k přidání daného kontroleru.

```
1  function pyString (str) {
2    if (!/^'.*'$/.test(str)) {
3      throw new TypeError(`Expected string, got: "${str}"`)
4    } else {
5      return str.substr(1, str.length - 2)
6    }
7  }
```

Fragment kódu 10: Převedení reprezentace hodnoty Pythonu z typu string na nativní datový typ JavaScriptu

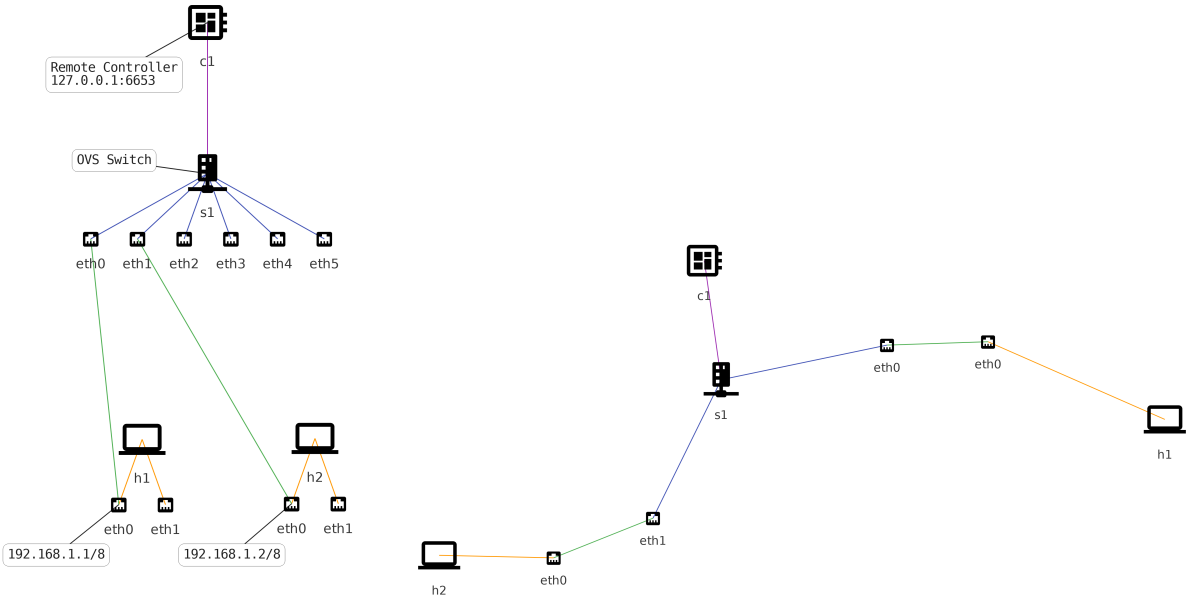
Po zpracování celého souboru jsou všechny nalezené položky pospojovány do funkční topologie. Výstupem je jednak export topologie ve formátu JSON (ukázka formátu

je k dispozici ve fragmentu kódu 5) a dále také log. V logu je popsáno vše, co selhalo, například pokud jsou nalezeny porty, které nejsou k ničemu připojeny, nebo pokud dojde k chybě při zpracovávání volání metody.

```
// Set up links (can't be done before all ports are in items)
links.forEach(edge => {
  edge.from = portMap[edge.from]
  edge.to = portMap[edge.to]
  if (edge.from && edge.to) {
    // Don't add edges to nowhere
    items.put(edge)
  }
})

return {
  log: log.map(v => ({ ...v, item: {} })),
  data: {
    ...jsonProps,
    version: 0,
    startScript: scriptLines.startScript.join('\n'),
    stopScript: scriptLines.stopScript.join('\n'),
    items: items.array
  }
}
```

Fragment kódu 11: Propojení portů a sestavení exportu

Jak je možné si všimnout na obrázku 4, v importovaném projektu chybí některé údaje a položky. Tou nejviditelnější změnou jsou polohy jednotlivých prvků, ty jsou při importu umístěny pomocí algoritmu, protože skript žádné informace o polohách jednotlivých prvků neobsahuje. V importované topologii také chybí porty, které před exportem nebyly připojeny linkem k žádnému jinému portu. To je způsobeno tím, že Mininet vytváří porty vždy v párech při spojování dvou prvků, a proto není možné vytvořit jeden samostatný port, který není k ničemu připojen. Jedinou výjimku tvoří fyzické porty, které odpovídají skutečným portů hostitelského počítače. Při importu jsou také ztraceny veškeré popisky a názvy linků, ani ty nejsou součástí skriptu a tudíž, pokud mají v topologii být, je nutné je po importu vytvořit ručně. Zachovány jsou nicméně



(a) Před importem **(b)** Po importu

Obrázek 4: Jednoduchá topologie s kontrolerem před a po importu ze skriptu.

všechna nastavení importovaných prvků (například IP adresy, typy jednotlivých prvků nebo skripty po spuštění) a linků (zde se jedná pouze o omezení provozu).

5.4.7 Export adresního plánu

Mininet editor umožňuje pro každou topologii exportovat adresní plán, který obsahuje adresy všech hostů v dané topologii. Výstupem je PDF dokument, který obsahuje název projektu a tabulku se třemi sloupci. V prvním sloupci je hostname, ve druhém název portu a ve třetím IP adresa. Všechny hosty i porty jsou v abecedním pořadí, IP adresy u každého portu jsou ponechány v takovém pořadí, v jakém jsou v projektu.

Hostname	Port	Address
h1	eth0	172.18.1.1/16
		fc00::1/32
h2	eth0	172.18.1.2/16
		fc00::2/32
h3	eth0	172.18.1.3/16
		fc00::3/32
	eth1	192.168.1.1/24

Obrázek 5: Ukázka tabulky adresního plánu

V kódu jsou z topologie vybrány všechny porty a následně je každý přiřazen ke svému hostu. Pro všechny porty i hosty jsou v tomto procesu spočteny součty IP adres, tyto součty jsou pak použity pro slučování buněk v prvních dvou sloupcích tak, aby ve výsledné tabulce byl každý hostname a každý port jen jednou a také aby vzniklo vizuálně přehledné členění. Takto zpracovaná struktura je následně, jak je ukázáno ve fragmentu kódu 12, převedena do jednoduchého pole. Použitá knihovna pak z obsahu tohoto pole, názvů sloupců a několika dalších nastavení připravý výslednou tabulku, která bude součástí vygenerovaného PDF dokumentu, včetně případného rozumného zalamování na více stránek. K této tabulce je ještě přidán nadpis s názvem projektu, ten je vložen i do metadat výsledného dokumentu. Poté už je PDF dokument jen sestaven a uložen na disk uživatele.

```
const body = []
Object.entries(this.plan)
.sort(compareEntries)
.forEach(([nodeHostname, node]) => {
  let firstInNode = true
  Object.entries(node.ports)
  .sort(compareEntries)
  .forEach(([portHostname, ips]) => {
    let firstInPort = true
    ips.forEach(ip => {
      const row = []

      if (firstInNode) {
        row.push({ rowSpan: node.length, content: nodeHostname,
          styles: { valign: 'middle' } })
      }
      if (firstInPort) {
        row.push({ rowSpan: ips.length, content: portHostname,
          styles: { valign: 'middle' } })
      }
      body.push([...row, ip])

      firstInNode = false
      firstInPort = false
```

```
25      })
26    })
27  })
```

Fragment kódu 12: Příprava řádků tabulky adresního plánu

5.4.8 Export obrázku

Pro export obrázků je k dispozici GUI s výběrem velikosti obrázku. Tu je možné přesně určit v pixelech anebo přibližně v centimetrech a to ve variantě pro zobrazení v počítači (cca. 96 DPI) a pro tisk (cca. 300 DPI). Po zvolení velikosti je obrázek vyrenderován a uložen ve formátu PNG. Tento formát je bezztrátový, díky čemuž nedochází ke vzniku artefaktů (jako například u formátu JPEG). Jistou nevýhodou bezztrátových formátů je relativně velká velikost výsledného souboru, ale vzhledem k povaze dat (obrázek typicky obsahuje rozsáhlé plochy bílé barvy, které se velmi dobře komprimují) mají výsledné soubory celkem příznivou velikost. Další výhodou je jeho nativní podpora v moderních webových prohlížečích a v naprosté většině dalšího softwaru.

Pro renderování obrázků je využita neinteraktivní komponenta plátna (více v kapitole 5.4.1), díky tomu obrázek vždy přesně odpovídá tomu, co je zobrazeno na plátně v aplikaci. Samotné renderování probíhá ve dvou pokusech.

V prvním pokusu je vyrendrována celá topologie naráz a okamžitě uložena jako PNG obrázek do počítače uživatele. Komponenta plátna je vložena do DOM, je ale umístěna tak, aby nebyla viditelná (z technického hlediska má její rodičovský element nulové rozměry a neumožňuje viditelné přetékání svých potomků), její velikost odpovídá požadované velikosti obrázku. Ve fragmentu kódu 13 je ukázáno, že topologie je vystředěna (zajišťuje funkce fit z knihovny Vis) a přiblížena (funkcí moveTo) do vhodné vzdálenosti tak, aby obrázek obsahoval topologii celou a zároveň aby došlo k vyplnění téměř celého obrázku (výsledný obrázek bude obsahovat určitý okraj). Poté je nastaven handler na ukončení vykreslování (událost afterDrawing emitovaná knihovnou Vis) a plátno je vykresleno (pomocí funkce redraw). Po vykreslení dojde k uložení stavu plátna do formátu PNG, ten je plátnem poskytnut jako blob (což je objekt obalující binární data, poskytuje například velikost v bajtech nebo typ binárních dat). Blob je následně uložen do počítače uživatele.

```
// Fit all items into the view and scale accordingly
this.net.fit()
this.net.moveTo({
  scale,
  animation: false
})

// Render image blob
const blob = await new Promise(resolve => {
  const handler = ctx => {
    this.net.off('afterDrawing', handler)
    ctx.canvas.toBlob(resolve, 'image/png')
  }
  this.net.on('afterDrawing', handler)
  this.net.redraw()
})
```

Fragment kódu 13: Napozicování topologie a export obrázku nativní metodou

Díky tomu, že tato metoda využívá nativní funkce prohlížeče, tak je velmi rychlá a má malé paměťové nároky. Může nicméně selhat, protože velikost plátna je v prohlížečích omezena, typicky existuje maximální šířka, výška a počet pixelů, který je obvykle podstatně menší než výška krát šířka. V extrémních případech tyto limity nemusí být dostačující ani pro tisk ve formátu A4 při 300 DPI (v dnešní době se to týká snad pouze mobilních zařízení). Nicméně i u prohlížečů s benevolentnějšími limity (otestovány Blink, Gecko a Webkit) může dojít k jejich překročení, zejména pokud je topologie opticky dělena na sekce prázdnými oblastmi a obrázek je určen pro tisk ve vysoké kvalitě. Limity ale není možné nijak zjistit (s výjimkou zvětšování plátna dokud nedojde k selhání), z tohoto důvodu je tato metoda vždy vyzkoušena a na záložní se přechází v případě, že selže. Nicméně vzhledem k tomu, že se limity u testovaných prohlížečů pohybují ve stovkách megapixelů, nemělo by za běžných okolností (při rozumně velkém obrázku) docházet k selhávání této metody.

Záložní metoda také využívá plátno, ale pouze o rozměru 1 000 krát 1 000 pixelů. Výsledný obrázek se vyrendruje ve formě dlaždic a ty se následně spojí do jednoho obrázku. Tato metoda má značné paměťové i výpočetní nároky, nicméně netrpí problémy s limity plátna, které obvykle nejsou nijak vázané na množství dostupné paměti, a proto

může být použita k jejich obejití a vyrenderování podstatně většího obrázku, než jaký umožňují limity plátna, má-li uživatel dostatečné množství operační paměti.

```
for (let row = 0; row < rows; ++row) {
  for (let col = 0; col < cols; ++col) {
    // Move the viewport
    this.net.moveTo({
      position: { x: 0, y: 0 },
      offset: {
        x: offset.x - tileSize * col,
        y: offset.y - tileSize * row
      },
      animation: false
    })
```

Fragment kódu 14: Posun viewportu při rendorování jednotlivých dlaždic obrázku

5.5 Jednotkové testování

Tato kapitola se zabývá jednotkovým testováním klíčových součástí Mininet editoru.

Jednotkové testy testují jednotlivé části aplikace (jednotky, jsou to ideálně jednotlivé funkce nebo metody) v izolaci od zbytku aplikace. V průběhu testu dostane testovaná jednotka předem dané vstupy a pak jsou kontrolovány její výstupy. To ověřuje správnou funkčnost jednotky případně její pozměněné verze nebo náhrady, při její výměně. Správně napsaný test zabrání nečekanému rozbití aplikace, a to zejména v jiných částech aplikace využívajících tuto jednotku, což nemusí být při provádění dané úpravy zjevné. Správná funkce testu záleží zejména na zvolených testovacích datech. Tato data musí ideálně pokrýt všechny možné případy, přitom je stačí pokrýt typově, není nutné a vetšinou ani realizovatelné otestovat všechny možné kombinace vstupů (jen jednoduché sečtení dvou celých čísel na 64bitovém počítači by vyžadovalo více než $3,4 \cdot 10^{38}$ různých vstupů k otestování tímto způsobem).

5.5.1 Export a import skriptů

Toto je pravděpodobně oblast aplikace, která je nejvíce náchylná na chyby, proto byla věnována značná pozornost jejímu testování, pro které byly zvoleny dvě odlišné metody. Tou první je porovnání výstupu exportu nebo importu s očekávaným výstupem. Druhou metodou je pak sekvence exportu, importu a reexportu a následné porovnávání jejich výstupů.

Pro otestování exportu je projekt exportován do skriptu a následně porovnán znak po znaku s očekávaným skriptem. Díky deterministické povaze exportu je pro stejný vstupní projekt vygenerován vždy stejný výstupní skript a je tedy možné použít tuto metodu. Nicméně test selže i při drobných změnách (například při změnách pořadí nebo formátování), které nijak neovlivňují jeho funkčnost, což vede k relativně častým úpravám testu při změnách v kódu exportu skriptů.

Obdobně je testován import skriptu. V tomto případě není výstupem textový řetězec, ale datová struktura. Díky tomu je možné zkontrolovat, že byly importovány očekávané počty položek jednotlivých typů a že importované položky mají očekávané vlastnosti, dále jsou také testována nastavení Mininetu. Položky je nicméně potřeba očistit od identifikátorů a pozic, které nejsou součástí skriptu, při importu jsou generovány nové a proto se budou lišit, to ale není na závadu.

```
describe('Item properties', () => {
  const cleanItems = getCleanItems(json.items)
  expectedItems.forEach(item => {
    it(`${item.type}/${item.hostname || '-'}`, () => {
      expect(cleanItems)
        .to.deep.include(item)
    })
  })
})
```

Fragment kódu 15: Test přítomnosti jednotlivých očekávaných položek

Druhá metoda testování se zabývá tím, jestli je import inverzní funkcí k exportu. Při tomto testu je brán v potaz fakt, že ne vše, co je součástí projektu, je i součástí skriptu (exportem jsou ztraceny například popisky, nezapojené porty, pozice prvků nebo je-

jich identifikátory), a také jsou ignorovány komentáře, jelikož nijak neovlivňují spustitelnost skriptu. Pro každý projekt je vyexportován skript, naimportován a znovu vyexportován. Původní projekt se následně porovnává s naimportovaným a stejně tak se porovnávají i oba vyexportované skripty. Tento test je schopen velmi dobře odhalovat nesoulad mezi importem a exportem skriptů.

```
describe('Items', () => {
  // Ports
  it('port', () => {
    const type = 'port'
    const typePl = 'ports'
    const items1 = getCleanItems(data1.items, type)
    const items2 = getCleanItems(data2.items, type)
    expect(items2, `The amount of ${typePl} differs.`)
      .to.have.lengthOf.at.most(items1.length)
    expect(items1, `Some ${typePl} were not imported correctly.`)
      .to.include.deep.members(items2)
  })

  // Other
  ;[
    { type: 'controller', typePl: 'controllers' },
    { type: 'host', typePl: 'hosts' },
    { type: 'link', typePl: 'links' },
    { type: 'switch', typePl: 'switches' }
  ].forEach(({ type, typePl }) => {
    it(type, () => {
      const items1 = getCleanItems(data1.items, type)
      const items2 = getCleanItems(data2.items, type)
      expect(items2, `The amount of ${typePl} differs.`)
        .to.have.lengthOf(items1.length)
      expect(items2, `Some ${typePl} were not imported correctly.`)
        .to.have.deep.members(items1)
    })
  })
})

it('script reexport', () => {
```

```
33    expect(
34      removeNonCode(script2),
35      'Script changed after importing and reexporting.'
36    ).to.equal(removeNonCode(script1))
37  })
```

Fragment kódu 16: Porovnání naimportovaných a originálních položek v testu

5.5.2 Úložiště

Testování úložiště je relativně snadná záležitost, je to dáno tím, že Vuex byl již od počátku programován tak, aby byl snadno testovatelný. Například mutace jsou prosté funkce, které přijímají stav úložiště jako první parametr a volitelně i druhý parametr s daty, která mohou být předána při volání. Stav úložiště je v aplikaci dost komplexní struktura pro zajištění reaktivity, nicméně veškerý kód funguje i na obyčejném objektu. Toho se dá využít pro snadné vytvoření potřebného stavu před testováním, reaktivita zde totiž není potřeba. Testování typicky probíhá ve třech krocích, prvním je vytvoření počátečního stavu úložiště, druhým zavolání testované funkce s vhodně zvolenými argumenty a posledním je kontrola stavu úložiště po provedení dané funkce.

Ve fragmentu kódu 17 je ukázán test změny v úložišti. Tento test ověřuje funkčnost všech typů úprav, tedy smazání existující položky, nahrazení existující položky položkou novou, aktualizace vlastností existující položky a přidání nové položky.

Nejprve je pomocí předpřipravené funkce `getMockStateWithTopo` (ta je napsána generically a je využita v mnoha testech) předpřipraven počáteční stav úložiště. Poté jsou z položek v topologii vybrány první čtyři. U první položky (`nA`) je změněn její identifikátor, tato položka bude v úložišti přepsána svou novou hodnotou, ale vzhledem k tomu, že byl její identifikátor změněn, tak bude přidána jako nová položka a původní verze zůstane zachována. Druhé položce (`nB`) byl změněn hostname, tato položka bude v úložišti taktéž přepsána, ale protože její identifikátor změněn nebyl, tak na rozdíl od té první zmizí její původní hodnota. Třetí položka (`uC`) má nadefinovány pouze identifikátor a hostname, u ni bude v úložišti provedena aktualizace původní hodnoty. Na čtvrté položce (`oD`) nebyly provedeny žádné operace, bude totiž z úložiště smazána.

V následující části je provedena změna v úložišti (`mutations.applyChange`) a následně je zkontrolováno, že změna proběhla korektně. Ověřeno je, že původní hodnota

položky A zůstala zachována v nezměněném stavu a to jak z hlediska reference tak i z hlediska hodnot všech jejich vlastností. Dále je ověřeno, že nová hodnota položky A byla přidána a že její reference skutečně ukazuje na přidávaný objekt. U položky B je ověřeno, že reference na původní hodnotu byla přepsána referencí na hodnotu novou. Položka C byla pouze aktualizována, proto je ověřeno, že reference v úložišti stále odkazuje na původní hodnotu a že její vlastnost hostname má hodnotu novou. Na konec se ověřuje neexistence položky D, která byla smazána.

```
const state = getMockStateWithTopo()

const originalValues = Object.values(state.data.items)
const [oA, oB, oC, oD] = originalValues
const oACopy = { ...oA }
const nA = {
  ...oA,
  id: 'A'
}
const nB = {
  ...oB,
  hostname: 'B'
}
const uC = {
  id: oC.id,
  hostname: 'C'
}

mutations.applyChange(state, {
  replace: [nA, nB],
  remove: [oD.id],
  update: [uC]
})

expect(state.data.items)
  .to.have.own.property(oA.id, oA, 'Original A should still be present')
  .that.deep.equals(oACopy, 'Original A shouldn\'t be changed in any way')
expect(state.data.items)
  .to.have.own.property(nA.id, nA, 'New A should be added')
```

```
30 expect(state.data.items)
31   .to.have.own.property(nB.id, nB, 'New B should replace original B')
32 expect(state.data.items)
33   .to.have.own.property(oC.id, oC,
34     'Original C should still be present (just altered)')
35   .that.has.own.property('hostname', uC.hostname,
36     'Updated C should have the new hostname')
37 expect(state.data.items)
38   .to.not.have.own.property(oD.id, oD, 'D shoudn\'t exist anymore')
```

Fragment kódu 17: Test uložení změn do úložiště

Ve fragmentu kódu 18 je možné vidět jednodušší test stejné mutace `applyChange`. Ta v tomto případě má do úložiště přidat (nebo nahradit pokud existuje, proto `replace`) položku, ta ale nemá žádné vlastnosti. Takováto položka by se do úložiště neměla nikdy dostat, místo toho by měla být vyhozena výjimka. Tento test se tedy dá přečíst jako expect function to throw nebo česky očekávejte, že funkce vyhodí výjimku.

```
1 const state = getMockStateWithTopo()
2
3 // Replace without id
4 expect(() => {
5   mutations.applyChange(state, {
6     replace: [{}]
7   })
8 }).to.throw()
```

Fragment kódu 18: Test uložení položky bez identifikátoru

5.6 Integrační testování

Cílem této kapitoly je objasnění zvoleného způsobu integračního testování Mininet editoru.

Integrační testování testuje aplikaci jako celek, jinými slovy testuje spolupráci všech součástí. Celý proces testování se v rámci možností blíží tomu, jak by aplikaci testoval člověk a to včetně způsobu navigování, klikání atd. Počítač ale není schopný vyhodnocovat obraz tak jako člověk a proto je nutné buďto klikat naslepo anebo lépe iden-

tifikovat jednotlivé elementy ve struktuře DOM. Identifikace elementů ve stránce je nicméně velkým problémem integračních testů, tyto problémy jsou zejména výrazné u SPA jako je právě Mininet editor. SPA jsou totiž velmi dynamické a za běhu mohou různě měnit takřka celý svůj DOM, kvůli tomu je velmi obtížné vyhledávat elementy, které je potřeba otestovat. Další problémy pak přináší možnost změn v kódu, které mohou velmi snadno rozbít testy, aniž by při tom způsobily nefunkčnost aplikace jako takové Ještě podstatně větší problémy přináší různé frameworky (v tomto případě Vue), transpilery (Babel) a bundlery (Webpack). Kvůli použití těchto nástrojů se vstupní kód napsaný programátorem (velké množství modulů s naformátovaným kódem v ES2018) dost odlišuje od kódu sestaveného projektu (jen pár souborů s minimalizovaným kódem v ES5). Díky tomu nemusí názvy tříd, elementů ale i další odpovídat tomu, co je napsáno ve zdrojových kódech, a tím značně ztížit orientaci ve stránce. Navíc se mohou různé identifikátory (jako například třídy nebo id) měnit při každém sestavení, což vylučuje jejich využití v testech.

V integračních testech napsaných pro Mininet editor slouží k nalezení jednotlivých elementů ve stránce speciální atributy (`data-cy`). Ty jsou přidané do aplikace jen a pouze pro testování a nemají žádný jiný účel. To zajišťuje, že jsou velmi stabilní a obvykle není důvod je nijak měnit. V případě, že se programátor přeci jen rozhodne takovýto atribut změnit, tak je mu okamžitě jasné, že bude muset upravovat i testy. Další výhodou tohoto přístupu je, že tyto atributy jsou voleny tak, aby spolu nekolidovaly. Každý takový identifikátor je obvykle unikátní a není tedy potřeba používat složité selektory, typicky stačí vyhledat jen tento identifikátor.

Pro organizaci testů se používá stejný princip seskupování (funkce `describe` a `it`) testů a stejné asserty (styly assert, should nebo expect) jako v jednotkových testech. Velký rozdíl je ale v tom, že Cypress nespouští testy ihned, při vykonávání příkazů je pouze registruje do fronty a až následně vykonává. To znamená, že kód testů, jehož části jsou ukázány ve fragmentech, se nejprve celý vykoná a až poté započne samotný test. Díky této architektuře je možné, aby byl celý test napsán jakoby všechen kód běžel synchronně a na testované stránce bylo vše okamžitě hotové a k dispozici pro testování. Čekání na události, na změny ve stránce atd. si zajišťuje Cypress automaticky sám (je možné to v případě potřeby i upravit).

5.6.1 Plátno

Integrační testování plátna není triviální záležitost, není totiž možné nijak zkontrolovat stav pixelů na plátně (Cypress při testech na plátno ani nic nerenderuje). Testování tedy probíhá naslepo a přítomnost prvků se ověřuje pomocí dvojkliku na místo, kde by měl daný prvek být, a zkontrolováním hodnot v inputech editačního dialogu. Ve fragmentu kódu 19 je ukázka testování plátna pomocí nástroje Cypress.

```
1  const itemPosition = { x: 150, y: 150 }
2
3  it('Place the item', () => {
4    cy.get('[data-cy=vis] canvas')
5      .meVisClick(itemPosition)
6  })
7
8  it('Save the item', () => {
9    cy.get(`[data-cy=edit-${type}]`)
10     .get('[data-cy=edit-save]')
11     .click()
12   cy.get(`[data-cy=edit-${type}]`)
13     .should('not.exist')
14 })
```

Fragment kódu 19: Umístění a uložení prvku v testu

Další problém se týká dvojkliku, jelikož Vis poslouchá pouze událost `pointerdown` a sám si určuje jestli se jedná o dvě nezávislá kliknutí nebo o dvojklik. Cypress ale implicitně provádí kontroly před každou událostí, tyto kontroly slouží k ověření, že by danou operaci mohl provést i člověk, například se testuje, jestli je element viditelný, jestli není překrytý jiným elementem nebo jestli není zakázaný (disabled). Při selhání těchto ověření selže i celý test (uživatel by nedokázal kliknout na element, který nevidí). Tyto kontroly nevadí při nezávislých událostech ani pro první kliknutí z dvojkliku, kde jsou i přínosné, například proto, že Cypress před kliknutím chvíli počká na zavření dialogu překrývajícího plátno, což lépe odpovídá chování lidí. Každý test ale trvá určitý čas, který není zanedbatelný, hlavním důvodem je analýza viditelnosti a překrývání elementů, která není triviální. Proto je nutné je pro druhou a všechny následující události

dvojkliku vypnout (pomocí volby `force: true`), aby se čas mezi jednotlivými událostmi na dostatečně zkrátil. Pokud jsou tyto kontroly zapnuté, tak se často stává (při testování se jednalo o přibližně 20 % případů), že zpoždění mezi jednotlivými událostmi je tak velké, že je Vis vyhodnotí jako dvě nezávislá kliknutí. S vypnutými kontrolami se tento problém v průběhu testování nevyskytl ani jednou, teoreticky nicméně nelze garantovat to, že všechny události budou vyhodnoceny tak, jak byly zamýšleny.

Ve fragmentu kódu 19 je možné si všimnout příkazu `.meVisClick`, ten byl vytvořen pro zjednodušení klikání na plátno. Umožňuje pomocí jednoho příkazu provést jednoduché nebo dvojité kliknutí libovolným tlačítkem myši na libovolné místo na plátně určené kartézskými souřadnicemi x a y. Část tohoto příkazu, která zajišťuje právě dvojklik, je ukázána ve fragmentu kódu 20.

```
1 cy.wrap(subject)
2   .trigger('pointerdown', { button, clientX: x, clientY: y })
3   .trigger('pointerup', { button, clientX: x, clientY: y, force: true })
4   .trigger('pointerdown', { button, clientX: x, clientY: y, force: true })
5   .trigger('pointerup', { button, clientX: x, clientY: y, force: true })
```

Fragment kódu 20: Implementace dvojkliku v integračních testech

5.6.2 GUI

Testování GUI je o poznání jednodušší. Cypress pro tyto účely poskytuje značné množství příkazů, nicméně bylo i tak nutné vzít v potaz určitá specifika. Jednak specifika použitého frameworku (Vue) a knihovny komponent (Vuetify), ale také vlastních komponent, vytvořených na míru potřebám Mininet editoru.

Ve fragmentu kódu 21 je možné vidět způsob zapsání hodnoty do textového pole, to může reprezentovat nejen text, ale i číselné hodnoty, práce s ním je nicméně stejná. Jedná se o část znovupoužitelného příkazu použitého napříč testy. Vyhledá ve stránce element podle zadaného klíče (`key`) a vymaže cokoli, co je v něm právě napsané. Následně do něho znak po znaku zapíše hodnoty předané v poli `values`, každá hodnota zde odpovídá jednomu řádku (proto jsou spojeny sekvencí `{enter}`, která v Cypressu vyvolá odřádkování).

```
cy.get(`[data-cy=${key}]`)
  .clear()
  .type(values.join('{enter}'))
```

Fragment kódu 21: Testování textových polí

Práce s ostatními typy vstupů využívá klikání pro změnu stavu. Například ve fragmentu kódu 22 je stav checkboxu měněn právě klikáním (podle předané hodnoty `clicks`, která odpovídá počtu kliknutí nezbytných pro změnu do požadovaného stavu). Zde je potřeba Cypress donutit (pomocí volby `force: true` při volání metody `click`) na tento checkbox kliknout, protože knihovna Vuetify nativní checkboxy používá pouze na pozadí a na popředí je překrývá svým vlastním elementem, ten nativní je ale zachován například pro lidi používající odečítací programy.

```
for (let i = 0; i < clicks; ++i) {
  cy.get(`[data-cy=${key}] input`)
    .click({ force: true }) // The input is hidden but works
}
```

Fragment kódu 22: Testování checkboxů

Testování zadaných hodnot pak probíhá jednoduchým vyhledáním elementu podle klíče a porovnání požadované hodnoty s hodnotou skutečně přítomnou. Ve fragmentu kódu 23 jsou řádky z proměnné `values` spojeny v textový řetězec, jenž by měl být i hodnotou testovaného elementu.

```
cy.get(`[data-cy=${key}]`)
  .should('have.value', values.join('\n'))
```

Fragment kódu 23: Ověření hodnoty v poli

6 Ukázkové úlohy

V rámci diplomové práce byly vypracovány ukázkové úlohy, které lze vypracovat za pomoci Mininet editoru. Každá úloha sestává ze dvou částí. Tou první je vytvoření topologie, její konfiguraci a exportování do skriptu, to vše v Mininet editoru. Druhou částí je pak ověření správné funkčnosti topologie při emulaci v Mininetu případně s využitím Ryu.

6.1 Příprava

Pro vypracování úloh je nezbytné mít k dispozici následující:

- Mininet editor (pro srovnání lze skripty napsat i ručně anebo vytvořit v MiniEditu),
- Mininet (pro spuštění samotné emulace),
- Open vSwitch (pro spuštění switchů),
- počítač s Linuxem (může být i virtuální),
- Ryu (případně jiný kontroler).

Mininet Na adrese http://mininet.org/download/ je k dispozici předpřipravený obraz pro virtuální stroj a zároveň i instrukce k instalaci přímo do operačního systému.

Před spuštěním Mininetu je potřeba nejprve spustit Open vSwitch Daemon.

```
1 sudo systemctl start ovs-vswitchd.service
```

Fragment kódu 24: Spuštění Open vSwitch Daemonu (systemd)

Samotný Mininet pak bude spuštěn ze skriptu vyexportovaného z Mininet editoru. Ten je nutné spustit s právy super uživatele (obvykle uživatel root), například ve složce se staženým skriptem:

```
1 sudo ./mininet_network.py
```

Fragment kódu 25: Spuštění vyexportovaného skriptu

Ryu Na adrese https://osrg.github.io/ryu/ se nachází návod na instalaci.

Pro potřeby všech úloh bude stačit kontroler simple_switch_13.py, který je standardní součástí Ryu. Kontrolery se v Ryu spouští pomocí manageru s cestou k aplikaci (Python skript obsahující samotnou logiku kontroleru). Spuštění kontroleru ze složky s Ryu nainstalovaným ze zdrojových kódů (obsahuje složku bin, ve které je soubor ryu-manager, a složku ryu/app, kde se nachází jednotlivé aplikace) lze provést následovně: Bez definování portu kontroleru se použije výchozí, tím je port 6653:

```
./bin/ryu-manager ryu/app/simple_switch_13.py
```

Fragment kódu 26: Spuštění Ryu simple switche (výchozí port)

S definováním portu kontroleru (umožňuje mimo jiného spustit více kontrolerů naráz), v tomto případě se jedná o starší port 6633:

```
./bin/ryu-manager ryu/app/simple_switch_13.py \
--ofp-tcp-listen-port 6633
```

Fragment kódu 27: Spuštění Ryu simple switche (port 6633)

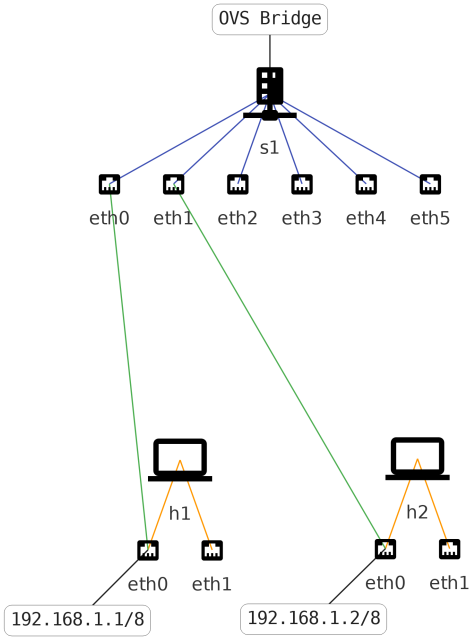
6.2 Jednoduchá topologie

Cílem této úlohy je spuštění jednoduché topologie bez kontroleru.

Kompletní adresní plán je k dispozici v příloze 1.

Sestavte síť v Mininet editoru podle přiloženého obrázku a nastavte IP adresy podle adresního plánu. Switch nastavte na OVS Bridge (ten nepotřebuje kontroler).

Po vyexportování a spuštění ověřte funkčnost spojení mezi 192.168.1.1 a 192.168.1.2. Pro ověření můžete použít příkaz `pingall`.



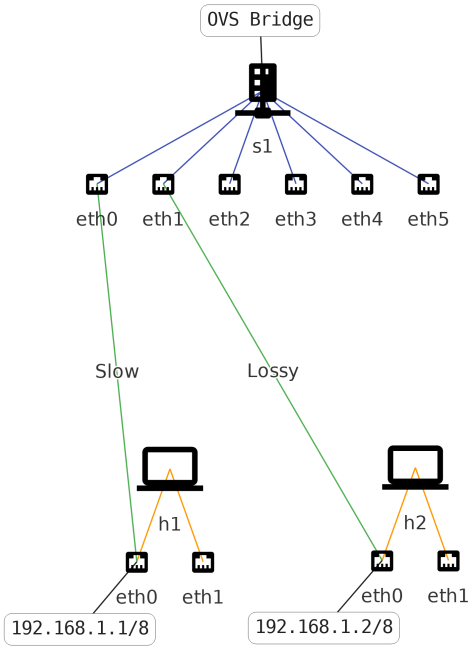
Obrázek 6: Jednoduchá topologie bez kontroleru.

6.3 Jednoduchá topologie s omezením provozu

Cílem této úlohy je vyzkoušet si omezování provozu v Mininetu.

Kompletní adresní plán je k dispozici v příloze 2.

Sestavte síť v Mininet editoru podle přiloženého obrázku a nastavte IP adresy podle adresního plánu. Switch nastavte na OVS Bridge (ten nepotřebuje kontroler). Na linku Slow nastavte omezení rychlosti (Bandwidth) na 10 Mibit/s, zpoždění (Delay) na 10 ms a jitter (Jitter) na 5 ms. Na linku Lossy nastavte ztrátovost (Loss) na 10 %.



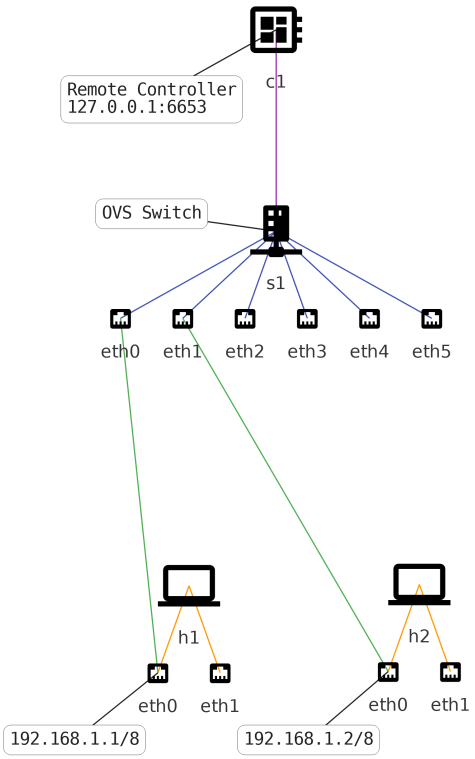
Obrázek 7: Jednoduchá topologie s omezením provozu bez kontroleru.

Po vyexportování a spuštění ověřte funkčnost spojení mezi 192.168.1.1 a 192.168.1.2. Zkontrolujte, že při komunikaci dochází ke ztrátám a ke zpoždění, které bylo nastaveno. Pro ověření můžete použít příkazy `h1 ping h2` a `h2 ping h1`. Příkaz `pingall` se nedá použít, protože posílá pouze jeden paket a nezobrazuje odezvu. Také změřte rychlost komunikace a ověřte, že nepřesahuje 10 Mibit/s. Pro změření rychlosti můžete použít příkaz `iperf h1 h2`. Rychlost bude pravděpodobně nižší kvůli zpoždění, jitteru a ztrátám.

6.4 Jednoduchá topologie s kontrolerem

Cílem této úlohy je propojení Mininetu s Ryu kontrolerem.

Kompletní adresní plán je k dispozici v příloze 3.

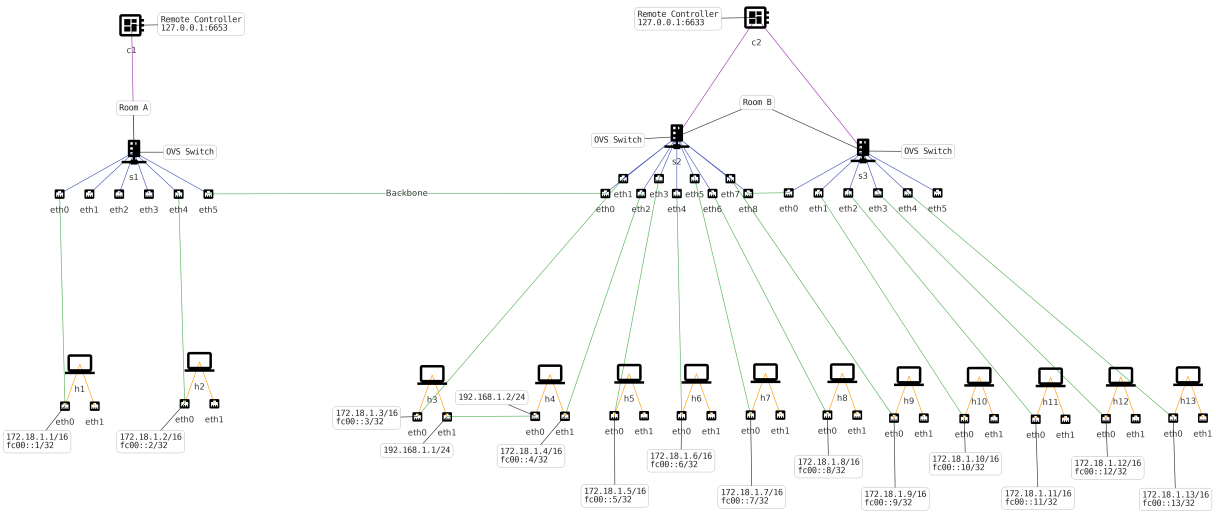


Obrázek 8: Jednoduchá topologie s kontrolerem.

Sestavte síť v Mininet editoru podle přiloženého obrázku a nastavte IP adresy podle adresního plánu. Pro kontroler ideálně použijte IP adresu 127.0.0.1 a port 6653, které jsou výchozí ve Ryu, pokud chcete použít jiné, musíte je změnit i v Ryu. Na rozdíl od předchozích úloh nastavte switch na OVS Switch, takto ho Ryu bude moci ovládat.

Po spuštění Ryu, vyexportování skriptu a jeho spuštění v Mininetu ověřte, že spojení mezi hosty s adresami 192.168.1.1 a 192.168.1.2 je funkční. Můžete využít příkaz `pingall`. Zkontrolujte, že ve výstupu z Mininetu a z Ryu vidíte, že došlo ke spojení. Pokud ke spojení nedojde, Mininet vypíše hlášku Unable to contact the remote controller at IP:port a nebude fungovat komunikace mezi hosty. Pokud ke spojení dojde, Ryu bude vypisovat zprávy začínající na paket in pro pakety přijaté na kontroleru a hosty budou schopny spolu komunikovat.

6.5 Komplexnější topologie s 2 kontrolery



Obrázek 9: Komplexnější topologie s 2 kontrolery.

Cílem této úlohy je spojení všeho, co jste si vyzkoušeli v předchozích úlohách, do jednoho většího projektu.

Kompletní adresní plán je k dispozici v příloze 4.

Sestavte síť v Mininet editoru podle přiloženého obrázku a nastavte IP adresy podle adresního plánu. Pro splnění úlohy není nezbytně nutné vytvářet hosty h5 až h12, stačí hosty h1, h2, h3, h4 a h13. Pro kontrolery je potřeba zvolit různé porty například standardní 6653 a historický 6633. Všem switchům nastavte typ na OVS Switch. U linku Backbone nastavte rychlost (Bandwidth) na 100 Mibit/s, zpoždění (Delay) na 10 ms, jitter na 5 ms a maximální délku fronty paketů (Max queue) na 42 paketů.

Po spuštění obou instancí Ryu, vyexportování skriptu a jeho spuštění v Mininetu ověřte funkčnost sítě. Pro ověření konektivity nepoužívejte příkaz `pingall`, neukáže vám totiž mezi kterými IP adresami komunikace probíhá, ani které linky jsou k tomu použity (v topologii existují dvě cesty mezi h3 a h4). Pro ověření konektivity je nutné využít příkaz `ping` případně `ping6` pro IPv6 v závislosti na vaší distribuci. Například příkazy `h3 ping 192.168.1.2` a `h4 ping 192.168.1.1` pro ověření přímého spojení mezi hosty h3 a h4. Každý host by měl být schopný komunikovat se všemi ostatními v IPv4 síti 172.18.1.0/16 a v IPv6 síti fc00::1/32, IPv4 síť 192.168.1.0/24 by měla být dostupná jen pro hosty h3 a h4. Ověřte i nedostupnost sítě 192.168.1.0/24 z ostatních hostů. Také otestujte rychlost komunikace pomocí příkazu `iperf` mezi různými částmi sítě. Komunikace by

měla být rychlá (v řádu Gbit/s, nicméně se hodně liší v závislosti na výkonu a vytížení počítače) pokud neprochází přes link Backbone (například h1 a h2, h3 a h4, h3 a h13) a jinak pomalá do 100 Mibit/s (například h2 a h3, h1 a h13). Odezva by měla být okolo 20 ms přes link Backbone a jinak téměř okamžitá (méně než 1 ms, ale přesná hodnota záleží na vašem systému).

Závěr

V teoretické části byly stručně popsány tradiční sítě a softwarově definované sítě. Byl zde popsán důvod vzniku softwarově definovaných sítí a jejich výhody v porovnání s tradičními sítěmi. Navíc byl popsán i vývoj protokolu OpenFlow, který dobře ilustruje rozvoj schopností softwarově definovaných sítí. Dále byl také uveden stručný výčet simulátorů a emulátorů včetně Mininetu, který byl popsán podrobněji i s návodem na použití.

Stěžejní částí této práce je aplikace Mininet editor. Tato aplikace byla úspěšně vytvořena a poskytuje veškerou funkcionalitu, která po ni byla požadována. Tedy vytvoření topologie, nastavení položek v ní a voleb emulace v Mininetu. Dále také export projektu pro pozdější import nebo ve formě skriptu pro emulaci v Mininetu, export obrázku topologie a adresního plánu. Součástí požadavků byl i import exportovaných projektů a to včetně importu skriptů, ten funguje, s přihlédnutím k povaze importovaných dat ve skriptech, velmi dobře. V rámci práce byl popsán návrh Mininet editoru, byly objasněny stěžejní části jeho kódu a bylo popsáno jeho testování. Byl také napsán popis MiniEditu, který byl inspirací pro Mininet editor, a porovnání obou aplikací.

Poslední částí práce jsou ukázkové úlohy z oblasti softwarově definovaných sítí. Byly vytvořeny celkem čtyři a v práci jsou seřazeny od nejjednodušší po nejkomplexnější. K jejich tvorbě byl využit Mininet editor, ten byl kromě otestování, že se úlohy skutečně dají vypracovat, použit pro vytvoření vložených obrázků topologií a také pro export tabulek s adresními plány, které jsou k dispozici v přílohách.

Použitá literatura

1. NATARAJAN, Sriram. *OpenFlow version 1.3 tutorial | SDN Hub* [online]. 2015 [cit. 2019-02-25]. Dostupné z: http://sdnhub.org/tutorials/openflow-1-3/.
2. CHING-HAO; LIN, Dr. Ying-Dar. *OpenFlow Version Roadmap* [online]. 2015 [cit. 2019-02-25]. Dostupné z: http://speed.cis.nctu.edu.tw/~ydlin/miscpub/indep_frank.pdf.
3. KERNER, Sean Michael. *ONF Announces Stratum Project to Redefine SDN* [online]. 2018 [cit. 2019-02-25]. Dostupné z: http://www.enterprisenetworkingplanet.com/netsp/openflow-is-the-past-as-onf-announcesstratum-project-to-redefine-sdn.html.
4. NSNAM. *About | ns-3* [online]. 2011-2019 [cit. 2019-02-21]. Dostupné z: https://www.nsnam.org/about/.
5. WETTE, P.; DRÄXLER, M.; SCHWABE, A.; WALLASCHEK, F.; ZAHRAEE, M. Hassan; KARL, H. *MaxiNet: Distributed Network Emulation* [online]. 2014 [cit. 2019-02-21]. Dostupné z: http://maxinet.github.io.
6. INC, EstiNet Technologies. *EstiNet X | Simulator* [online]. 2019 [cit. 2019-02-21]. Dostupné z: http://www.estinet.com/ns/?page_id=21140.
7. CISCO SYSTEMS, Inc. *VIRL Micro-Site* [online]. 2018 [cit. 2019-02-21]. Dostupné z: http://virl.cisco.com.
8. TEAM, Mininet. *Mininet Overview - Mininet* [online]. 2018 [cit. 2019-01-15]. Dostupné z: http://mininet.org/overview/.
9. LANTZ, Bob; GEE, Gregory. *mininet/miniedit.py at master · mininet/mininet* [online]. 2018 [cit. 2019-02-18]. Dostupné z: https://github.com/mininet/mininet/blob/master/examples/miniedit.py.
10. *JavaScript®*. Redwood Shores, CA, USA, 1995.
11. ECMA INTERNATIONAL. ECMAScript® 2018 Language Specification. In: 9. vyd. Rue du Rhone 114, CH-1204 Geneva, 2018, kap. Introduction, s. 43–44.

12. YOU, Yuxi (Evan). *Vue.js is a progressive, incrementally-adoptable JavaScript framework for building UI on the web.* [online]. 2013-2019 [cit. 2019-01-15]. Dostupné z: https://github.com/vuejs/vue.

13. YOU, Yuxi (Evan). *What is Vuex? | Vuex* [online]. 2017 [cit. 2019-01-15]. Dostupné z: https://vuex.vuejs.org/.

14. LEIDER, John. *Material Component Framework for Vue.js 2* [online]. 2018 [cit. 2019-01-15]. Dostupné z: https://github.com/vuetifyjs/vuetify.

15. B.V., Almende. *vis.js - A dynamic, browser based visualization library.* [online]. 2010-2017 [cit. 2019-01-15]. Dostupné z: http://visjs.org/.

16. ANDREWS, Austin. *Material Design Icons* [online]. 2018 [cit. 2019-01-14]. Dostupné z: https://materialdesignicons.com/.

17. PARR, Terence. *ANTLR* [online]. 2014 [cit. 2019-01-14]. Dostupné z: https://www.antlr.org/.

18. HALL, James. *Client-side JavaScript PDF generation for everyone. https://parall.ax-/products/jspdf* [online]. 2019 [cit. 2019-02-13]. Dostupné z: https://github.com/MrRio/jsPDF.

19. KOPPERS, Tobias. *A bundler for javascript and friends. Packs many modules into a few bundled assets. Code Splitting allows for loading parts of the application on demand. Through "loaders", modules can be CommonJs, AMD, ES6 modules, CSS, Images, JSON, Coffeescript, LESS, ... and your custom stuff.* [online]. 2018 [cit. 2019-01-15]. Dostupné z: https://github.com/webpack/webpack.

20. TEAM, Babel. *Babel is a compiler for writing next generation JavaScript.* [online]. 2019 [cit. 2019-01-15]. Dostupné z: https://github.com/babel/babel.

21. *Mocha - the fun, simple, flexible JavaScript test framework* [online]. 2019 [cit. 2019-01-15]. Dostupné z: https://mochajs.org/.

22. *Chai* [online]. 2018 [cit. 2019-01-15]. Dostupné z: https://www.chaijs.com/.

23. CYPRESS.IO. *Why Cypress? | Cypress Documentation* [online]. 2018 [cit. 2019-01-15]. Dostupné z: https://docs.cypress.io/guides/overview/why-cypress.html#In-a-nutshell.

Seznam příloh

Adresní plán - Jednoduchá topologie

Hostname	Port	Address
h1	eth0	192.168.1.1/8
h2	eth0	192.168.1.2/8

Adresní plán - Jednoduchá topologie s omezením provozu

Hostname	Port	Address
h1	eth0	192.168.1.1/8
h2	eth0	192.168.1.2/8

Adresní plán - Jednoduchá topologie s kontrolerem

Hostname	Port	Address
h1	eth0	192.168.1.1/8
h2	eth0	192.168.1.2/8

Adresní plán - Komplexnější topologie se 2 kontrolery

Hostname	Port	Address
h1	eth0	172.18.1.1/16
		fc00::1/32
h2	eth0	172.18.1.2/16
		fc00::2/32
h3	eth0	172.18.1.3/16
		fc00::3/32
	eth1	192.168.1.1/24
h4	eth0	192.168.1.2/24
	eth1	172.18.1.4/16
		fc00::4/32
h5	eth0	172.18.1.5/16
		fc00::5/32
h6	eth0	172.18.1.6/16
		fc00::6/32
h7	eth0	172.18.1.7/16
		fc00::7/32
h8	eth0	172.18.1.8/16
		fc00::8/32
h9	eth0	172.18.1.9/16
		fc00::9/32
h10	eth0	172.18.1.10/16
		fc00::10/32
h11	eth0	172.18.1.11/16
		fc00::11/32
h12	eth0	172.18.1.12/16
		fc00::12/32
h13	eth0	172.18.1.13/16
		fc00::13/32