Mendelova univerzita v Brně
Provozně ekonomická fakulta

Vizualizace komunikačního procesu v počítačové síti

Bakalářská práce

Vedoucí práce:
Ing. Jiří Balej

Martin Süss

Brno 2017

Tímto chci poděkovat Ing. Jiřímu Balejovi za ochotu pomoci a zejména za poskytnuté zázemí a prostředky v době tvorby práce. Také děkuji za poskytnuté konzultace s Ing. Janem Kolomazníkem, Ph.D., které mi velmi pomohly při tvorbě práce a bádání. Děkuji také své rodině a kamarádům, kteří mě po celou dobu podporovali, dodávali motivaci a sílu.

Čestné prohlášení

Prohlašuji, že jsem tuto práci: **Vizualizace komunikačního procesu v počítačové síti**
vypracoval samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*
Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.
Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 18. května 2017 ....

Abstract

Süss, M. Visualization of the communication process in Computer network. Bachelor thesis. Mendel University in Brno, 2017.

This bachelor thesis deals with the creation of an application for visualization of the communication process between a certain source and destination device. The goal of the application is to give the user an idea of how the network „looks" between these two devices and possibly simplify troubleshooting of the network. It allows the user to see which network devices are in that path, and also provide basic configuration information for each of them. In case of unsuccessful communication, it can tell basic error information.

Keywords: Computer network, device, path search, visualization, troubleshooting.

Abstrakt

Süss, M. Vizualizace komunikačního procesu v počítačové síti. Bakalářská práce. Mendelova univerzita v Brně, 2017.

Tato bakalářská práce se zabývá vytvořením aplikace pro vizualizaci komunikačního procesu mezi určitým zdrojovým a cílovým zařízením. Cílem aplikace je uživateli dodat představu o tom, jak síť mezi těmito dvěma zařízeními „vypadá" a případně zjednodušit odstraňování problémů (tzv. troubleshooting) v síti. Umožňuje uživateli vidět, které síťové prvky se v dané cestě nacházejí a u každého z nich také sdělit základní informace o konfiguraci. V případě neúspěšné komunikace dokáže říci základní informace o chybě.

Klíčová slova: Počítačová síť, zařízení, hledání cesty, vizualizace, odstraňování problémů.

Obsah

1 Úvod **15**
1.1 Uvedení do problematiky 15
1.2 Troubleshooting síťové komunikace 15

2 Cíl práce a metodika **17**
2.1 Cíl práce 17
2.2 Metodika 17

3 Analýza současného stavu **19**
3.1 Požadavky na aplikaci 19
3.2 Informační zdroje 19
3.3 Klíčová slova 20
3.4 Výsledky hledání v závěrečných pracích 21
3.4.1 Vizualizace síťového provozu – Martin Matoušek (2014) . . . 21
3.4.2 Vizualizace stavu sítě – Bc. Martin Hejna (2010) 21
3.4.3 Uživatelské rozhraní pro vizualizaci rozsáhlých sítí – Tomáš Tauer (2013) 21
3.4.4 Vizualizácia siet'ových topológií – Milan Pánik (2014) 22
3.5 Automatické nástroje nalezené v závěrečné práci 22
3.5.1 10strike LANState 22
3.5.2 NetDepict 23
3.5.3 NTM 23
3.6 Výsledky hledání pomocí vyhledávače Google 24
3.6.1 10scape 24
3.6.2 Cytoscape 24
3.6.3 TNV (The Network Visualizer) 25
3.6.4 LiveAction 25
3.7 Shrnutí 26

4 Použité technologie **27**
4.1 Protokoly pro vzdálený přístup k síťovým zařízením 27
4.1.1 Telnet 27
4.1.2 SSH 27
4.1.3 SNMP 28
4.2 Programovací jazyk JAVA 28
4.2.1 Obecná charakteristika 28
4.2.2 Knihovny 29
4.3 Knihovna jQuery 29
4.4 Neo4J databáze 30
4.4.1 Úvod 30
4.4.2 Dotazovací jazyk *Cypher* 31
4.5 In-memory databáze 32

5 Návrh aplikace 33
5.1 Momentální stav existujícího projektu . . . 33
5.1.1 Frontend . . . 33
5.1.2 Backend . . . 33
5.1.3 Upravený návrh backendu – část CORE . . . 35
5.1.4 Upravený návrh backendu – část API . . . 38
5.2 Specifikace vyvíjené aplikační části . . . 41
5.3 Funkční stránka vyvíjené aplikační části . . . 42
5.4 Backend . . . 43
5.4.1 Objektový návrh . . . 43
5.4.2 Návrh grafové databáze . . . 47
5.4.3 Návrh algoritmu pro nalezení cesty . . . 47
5.5 Frontend . . . 50
5.6 Testovací síť . . . 52

6 Implementace aplikace 53
6.1 Implementace úprav části CORE v existujícím projektu . . . 53
6.1.1 Databázové třídy . . . 53
6.1.2 Třídy implementující služby . . . 54
6.1.3 Repositářové třídy . . . 55
6.2 Implementace úprav části API v existujícím projektu . . . 55
6.2.1 Třídy typu Kontrolér . . . 56
6.2.2 Třídy typu Zdroj . . . 57
6.2.3 Třídy typu Překladač zdrojů . . . 58
6.3 Implementace hledání cesty . . . 59
6.3.1 Vzdálený přístup k zařízení . . . 59
6.3.2 Komunikátor a jeho potomci . . . 61
6.3.3 Parsování dat . . . 62
6.3.4 Nalezení cesty . . . 63
6.4 Frontend . . . 65
6.4.1 HTML soubor . . . 65
6.4.2 CSS soubor . . . 66
6.4.3 JavaScript soubor . . . 67
6.5 Objevené problémy při implementaci . . . 69
6.5.1 ARP tabulka . . . 69
6.5.2 Hostname zařízení . . . 70

7 Testování aplikace 71
7.1 Topologie . . . 71
7.2 Popis konfigurace . . . 71
7.3 Scénáře bez problému . . . 74
7.3.1 První scénář . . . 74
7.3.2 Druhý scénář . . . 74

7.3.3 Třetí scénář 74
7.3.4 Čtvrtý scénář 74
7.4 Scénáře s problémem 77
7.4.1 První scénář 77
7.4.2 Druhý scénář 77
7.4.3 Třetí scénář 77
7.4.4 Čtvrtý scénář 78

8 Diskuse a závěr **81**
8.1 Omezení a předpoklady 82
8.2 Možné návrhy na vylepšení aplikace do budoucna 83

9 Reference **85**

Přílohy **89**

A Seznam zkratek **90**

B Elektronická příloha **91**

Seznam obrázků

Obrázek 1: Propojení backendových částí API a CORE 34

Obrázek 2: UML návrh změn v části CORE – 1. část 36

Obrázek 3: UML návrh změn v části CORE – 2. část 37

Obrázek 4: UML návrh změn v části API – 1. část 39

Obrázek 5: UML návrh změn v části API – 2. část 40

Obrázek 6: Vývojový diagram vyvíjené aplikace 42

Obrázek 7: Objektový návrh části hledající cestu 46

Obrázek 8: Demonstrativní síť algoritmu pro nalezení cesty 49

Obrázek 9: Grafický návrh frontendu s úspěšným nalezením cesty 50

Obrázek 10: Grafický návrh frontendu s neúspěšným nalezením cesty 51

Obrázek 11: Návrh testovací sítě pro účely implementace 52

Obrázek 12: Ilustrativní příklad dialogového okna 69

Obrázek 13: Komplexní topologie pro testování 72

Obrázek 14: Výsledek prvního bezproblémového scénáře 75

Obrázek 15: Výsledek druhého bezproblémového scénáře 75

Obrázek 16: Výsledek třetího bezproblémového scénáře 76

Obrázek 17: Výsledek čtvrtého bezproblémového scénáře 76

Obrázek 18: Ukázka příkazu ping pro obnovu tabulek 76

Obrázek 19: Výsledek prvního scénáře s problémem 78

Obrázek 20: Výsledek druhého scénáře s problémem 78

Obrázek 21: Výsledek třetího scénáře s problémem 79

Obrázek 22: Výsledek čtvrtého scénáře s problémem 79

1 Úvod

1.1 Uvedení do problematiky

Slovní spojení **počítačové sítě** už skoro každý člověk zná. Je to tím, že všichni jsme součástí nějaké počítačové sítě, ať už doma nebo v zaměstnání. Nicméně specifické odvětví počítačových sítí je tzv. *troubleshooting*, což je podle *Techopedie* (2016) „kolektivní opatření a procesy používané k identifikaci, diagnostice a řešení problémů a chyb v rámci počítačové sítě. Jedná se o systematický proces, který si klade za cíl vyřešit problémy a obnovit normální stav počítačové sítě."

Typickým denním problémem je nemožnost spojit se z určitého bodu A do bodu B. V malých sítích je řešení otázkou pár minut pro zkušenější „síťaře", nicméně čím větší je síť (resp. počet síťových prvků a kroků mezi bodem A a bodem B), tím se zvětšuje (někdy až neúměrně růstu sítě) čas řešení problému. Podle Martina Matouška (2014, str. 3) se v dnešní době síťové toky (a tedy i počet zařízení) zvětšují a dorostly do takové míry, že jejich monitorování vytváří obrovské množství statistik a informací, ve kterých se dnešní správci sítě musí rychle orientovat.

A zde začíná mít uplatnění logická vizualizace sítě, která troubleshooting zjednoduší tím, že řešitel problému přesně ví, které prvky jsou mezi bodem A a bodem B, a tedy také ví, které prvky budou s největší pravděpodobností problémové. Bez takové vizualizace by bylo nutné krok po kroku zjišťovat, které prvky leží v cestě, a to opakovaně pro každý jednotlivý problém (pokud by se nejednalo o stejnou nebo velmi podobnou cestu), což je, jak je jistě zřejmé, velmi časově neefektivní.

1.2 Troubleshooting síťové komunikace

Vzhledem k tomu, že v práci je často zmiňován pojem troubleshooting, je dobré si hned na začátku podrobněji vysvětlit, o co vlastně jde a jaký to má vztah k vyvíjené aplikaci. Pojem troubleshooting lze do češtiny přeložit jako „hledání závady" či „řešení problémů", což je přesně to, co se pod tímto pojmem má chápat.

Troubleshooting počítačové sítě probíhá iterativním způsobem – postupuje se od počátečního prvku postupně až ke koncovému a zjišťuje se, kde je chyba. Ta se dá většinou odhalit tak, že u každého prvku se zkontroluje konfigurace a stav. Např. u zařízení společnosti Cisco na to slouží tzv. *show* [1] příkazy v příkazovém řádku (Odom, 2013). Díky nim si uživatel může nechat vypsat aktuální nastavení takřka čehokoliv, dále různé stavy, počítadla a mnohé další.

Avšak procházet prvky od jednoho k dalšímu je zdlouhavá činnost, zejména pak, když problémový prvek je až jeden z posledních v cestě. A tohle by měla usnadnit a urychlit vyvíjená aplikace, která vizualizuje prvky tak, jak jsou za sebou postupně zapojeny v cestě z bodu A do bodu B. Tím do jisté míry odpadá nutnost začít s troubleshootingem od samotného začátku, ale je možné začít řešit chybu až

[1] „Show" proto, že všechny příkazy, týkající se zobrazení určitého nastavení či konfigurace, začínají právě tímto klíčovým slovem.

u prvku, který aplikace označí za „chybový“, což potenciálně může správcům sítě ušetřit spoustu času. Dá se tedy říci, že ulehčení troubleshootingu správcům sítě je jedna z největších motivací pro tuto práci.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem práce je na základě požadavků Ústavu informačních technologií Mendelovy univerzity v Brně navrhnout, implementovat a otestovat aplikaci, která bude umožňovat zmíněnou vizualizaci síťových prvků v počítačové síti, ležících v cestě z určitého zdrojového bodu k bodu koncovému. Aplikace bude také nabízet výpis jak stručných konfiguračních informacích o každém zařízení, tak i případného problému, zamezujícího nenalezení kompletní cesty.

Aplikace by měla směřovat do rukou správců sítě, tudíž by měla být uživatelsky přívětivá, dále by měla umožňovat zakomponování do již existujícího projektu, který se zabývá správou síťové infrastruktury, resp. jej obohatit o tuto funkcionalitu.

2.2 Metodika

Splnění stanovených cílů práce vyžaduje vykonání několika nezbytných kroků. Bude nezbytné prozkoumat a nastudovat určité oblasti, analyzovat současný stav, navrhnout vlastní postupy a ty následně implementovat do podoby počítačového kódu.

Nejdříve se stanoví požadavky na aplikaci. Měly by být konkrétní a přesné, neboť na základě nich se odvíjí všechny další kroky. Poté se provede analýza současného stavu aplikací, které se zabývají vymezenou problematikou. Výsledek analýzy odpoví na otázku, zdali existuje řešení, které splňuje vytyčené požadavky.

V případě, že v současném stavu nebude existovat přípustné řešení, začne se pracovat na nové aplikaci, která zaplní nalezené „bílé místo“. Nejdříve se určí technologie a prostředky, kterými bude realizován vývoj. Stanoví se programovací jazyky, technologie pro ukládání dat, prostředky pro komunikaci se zařízeními atd.

Tvorba nové aplikace se bude skládat ze tří postupných kroků. První krok – návrh – „připraví půdu“ pro další krok a stanoví, jak bude aplikace vypadat. Druhý krok – implementace – převede myšlenky, postupy a principy, navržené v předešlém kroku, do programového kódu. Poslední krok – testování – ověří, zdali aplikace funguje tak, jak se předpokládá.

Návrh bude rozdělen do dvou etap. Vzhledem k tomu, že existuje projekt ve vývoji, bude v první etapě nutné prozkoumat, v jakém je stavu a co bude nezbytné doplnit pro potřeby nového řešení.

Ve druhé etapě se bude pokračovat v návrhu vlastní části aplikace, zodpovědné za nově přinášenou funkcionalitu – vizualizaci komunikačního procesu. Součástí bude objektový návrh, návrh databází či algoritmu pro nalezení cesty, který lze uskutečnit až v případě nastudování nezbytných principů[2].

Poté se práce přesune na část implementační. Ta celý návrh zhmotní do programového kódu, čímž se vytvoří spustitelná aplikace. Její testování bude probíhat

[2]Bude nezbytné nastudovat principy fungování síťových zařízení, které určují, jak se hledá správná cesta do cílového bodu.

nasazením do reálného prostředí. Navrhnou se odlišné scénáře, které aplikaci otestují v různých situacích, jež mohou v produkční sféře nastat. Pokud testování dopadne úspěšně, bude to značit správnost návrhu i implementace.

3 Analýza současného stavu

3.1 Požadavky na aplikaci

Má-li být aplikace přínosná a použitelná, je třeba, aby splňovala řadu velmi specifických požadavků. Měla by, i na základě společné diskuze s pracovníky Ústavu informačních technologií Mendelovy univerzity v Brně, splňovat tyto požadavky:

- být uživatelsky přívětivá,
- automaticky vizualizovat všechny síťové prvky, ležící v cestě z bodu A do B,
- znázorňovat základní důležité informace o každém zařízení, jako jsou například:
  - stavy rozhraní,
  - tabulky (CAM tabulka, směrovací tabulka, ARP tabulka),
  - VLAN informace,
- v místě problému znázornit prvek a vypsat pravděpodobnou chybu,
- být zadarmo nebo za přijatelnou cenu (v rámci tisíců Kč),
- umět vizualizovat střední až velké sítě (tj. alespoň desítky prvků),
- umožňovat integraci do komplexnějšího řešení správy sítě (integrovat se jako modul),
- umět se spojit s prvky pomocí protokolu *Telnet*, popřípadě také pomocí protokolu *SSH*.

3.2 Informační zdroje

Hledání již existujícího řešení bude prováděno jak v odborných závěrečných pracích, tak i komerčních či open source aplikacích.

Jako informační zdroj závěrečných prací budou brány portál Masarykovy Univerzity (2016-a), Mendelovy Univerzity (2016-b) a VUT (2016-c) pro vyhledávání v závěrečných pracích, dále webové stránky *Thesis.cz* (2016), které slouží jako globální vyhledávač prací pro velkou část vysokých škol v ČR. Spolehlivost těchto stránek je velmi vysoká, neboť se jedná o oficiální stránky vysokých škol a veřejných institucí. Všechny tyto nástroje umí vyhledávat na základě klíčových slov. Práce musí být maximálně 5 let stará, ostatní budou považovány za zastaralé a nepoužitelné, neboť 5 let je v branži IT velmi dlouhá doba. Spolehlivost prací bude prokazována na základě úspěšného obhájení.

Dále budou aplikace hledány pomocí nejrozšířenějšího vyhledávače *Google* (2016), který je velmi nestranným (co se reklam týče) a spolehlivým vyhledávačem. Zdroje či aplikace musí být opět maximálně 5 let staré. Spolehlivost vyhledaných

řešení bude zjišťována na základě referencí a propracovanosti webových stránek i samotného programu, což samozřejmě může být ovlivněno subjektivním názorem.

3.3 Klíčová slova

Vzhledem k tomu, že je převážná většina „světa IT“ psaná a dělaná v anglickém jazyce, i výrazy pro hledání aplikace budou, vedle českého jazyka, v tomto jazyce.

Jako nejadekvátnější klíčová slova budou zvolena tato:

- **v češtině**: síť, vizualizace, topologie, průchod, paket, datagram, data, provoz, reálný čas, komunikace, síťový tok,
- **v angličtině**: network, visualization, topology, net flow, packet, datagram, data, traffic, real-time, communication, network flow.

Z těchto klíčových slov potom vzniknou např. tato slovní spojení:

- **v češtině**:
  - vizualizace komunikačního procesu v síti,
  - vizualizace počítačové sítě,
  - vizualizace síťové topologie,
  - vizualizace průchodu paketu | datagramu | dat,
  - vizualizace síťového provozu,
  - vizualizace toku sítě,
  - vizualizace sítě v reálném čase,
- **v angličtině**:
  - network communication visualization,
  - network visualization,
  - network topology visualization,
  - packet throughput visualization,
  - traffic visualization,
  - network flow visualization,
  - real-time network visualization.

3.4 Výsledky hledání v závěrečných pracích

Budou zde postupně předloženy závěrečné práce, které souvisejí se zkoumanou problematikou a přinášejí zajímavé poznatky či řešení. Hledání proběhlo na zmíněných portálech za pomoci vymezených klíčových slov. V každé sekci se práce nejdříve představí a poté se zhodnotí její přínos.

3.4.1 Vizualizace síťového provozu – Martin Matoušek (2014)

Práce byla nalezena na portálu pro hledání závěrečných prací VUT v Brně a zároveň to byla jediná práce z této VŠ, která spadá do oblasti hledání cesty v síti. Teoretická část práce pojednává o technologii *NetFlow*[3]. Praktická část práce je zaměřena na návrh a implementaci aplikace na vizualizaci síťového provozu. Pro účely rešerše je zajímavá především praktická část.

Aplikace získává data pro vizualizaci z již zmíněné technologie *NetFlow*. Na základě filtrů umožňuje vizualizovat jen to, co uživatel opravdu chce (na základě IP adresy, času atd.). Zobrazuje určitý tok a jeho základní informace, dále prvky a jejich vzájemné propojení.

Hlavním nedostatkem této aplikace vzhledem k výše zmíněným požadavkům je, že aplikace zobrazuje pouze koncové uzly (resp. všechny možné toky mezi různými koncovými uzly), nikoliv však jednotlivé prvky mezi nimi. Tudíž odpadá jakýkoliv troubleshooting, neboť nevidíme, který prvek je tím posledním dosažitelným v případě problému a také nemůžeme zjišťovat informace o žádném z prvku, což bylo dalším požadavkem. Dále je nežádoucí získávat data pomocí technologie *NetFlow*, neboť se zjistí informace pouze z pohledu koncových uzlů, navíc požadovány jsou protokoly *Telnet* či *SSH*. Aplikace je funkční, elegantně navržená a uživatelsky přívětivá, nicméně nesplňuje více požadavků, tudíž se jeví jako nevyhovující.

3.4.2 Vizualizace stavu sítě – Bc. Martin Hejna (2010)

Práce pochází z VŠE v Praze, konkrétně pak z fakulty informatiky a statistiky. Byť je práce starší než 5 let, je velmi pěkně zpracovaná. Výstupem práce sice není funkční aplikace sloužící pro vizualizaci sítě, nicméně velká část práce pojednává o principech vizualizace sítě, vymezení důležitých termínů, metod, metrik, přínosů či rizik. Pro budoucí vývoj aplikace by tedy práce mohla sloužit k ujasnění si některých souvislostí.

3.4.3 Uživatelské rozhraní pro vizualizaci rozsáhlých sítí – Tomáš Tauer (2013)

Aplikace se zaměřuje na vizualizaci komplexních sítí tak, aby byly i přes velké množství uzlů přehledné. Byť aplikace vykresluje i síťové prvky mezi koncovými uzly, opět

[3] Jedná se o proprietární technologii od společnosti Cisco, sloužící pro sběr dat ze síťového toku.

zcela chybí možnost zjistit podrobné informace o jednotlivém síťovém prvku. K dispozici jsou pouze základní informace, jako je IP adresa, MAC adresa. Autor neřeší získávání dat ze síťových prvků, pouze jejich zobrazení.

Data získává společnost *SolarWinds Czech s. r. o.*, která s autorem spolupracuje, prostřednictvím *SNMP* protokolu. Algoritmus na získávání dat je tím pádem samozřejmě know-how firmy, tudíž se vylučuje použitelnost aplikace pro splnění našeho cíle. Mimo to aplikace nesplňuje více bodů, nicméně tohle je stěžejní bod, který se nedá přehlížet či obejít.

3.4.4 Vizualizácia sieťových topológií – Milan Pánik (2014)

Bakalářská práce z roku 2014 od studenta Masarykovy univerzity je trochu jiná než ostatní zmíněné práce. Má totiž velmi významnou část rešeršní, ve které pojednává o nástrojích vizualizace síťové topologie. Práce je velmi přínosnou, neboť řeší přesně to, co si dává za cíl objevit tato rešerše. Významná část práce se zaobírá analýzou současného stavu aplikací zabývajících se vizualizací sítě.

Pánik (2014, str. 12) rozděluje nástroje do dvou kategorií – manuální a automatické. Manuální nástroje vyžadují interakci uživatele a přinášejí určité ulehčení při návrhu či kreslení sítě (Pánik, 2014, str. 11). Nesplňují však požadavky na aplikaci, která má být ve vykreslování sítě výhradně soběstačná a dynamická.

O automatických nástrojích pojednává samostatná kapitola 3.5.

3.5 Automatické nástroje nalezené v závěrečné práci

Jak již bylo zmíněno v sekci 3.4.4, závěrečná práce Milana Pánika (2014) rozděluje nástroje pro vizualizaci síťové topologie na manuální a automatické. Automatické nástroje jsou schopné vygenerovat celou infrastrukturu sítě samy. Autor dále rozděluje automatické nástroje s automatickou detekcí sítě a bez automatické detekce sítě (resp. vizualizace sítě na základě importovaného souboru). Jako přijatelné se jeví aplikace s automatickou detekcí sítě, a proto budou vyhovující z nich postupně probrány. Vzhledem k počtu těchto nástrojů je jim vyhrazena celá tato kapitola.

3.5.1 10strike LANState

Jedná se o velmi jednoduše ovladatelný program, který uživatele navádí krok po kroku, jak vizualizovat svou síť. Nejdříve naskenuje síť a poté prvky rozhází po kreslící ploše. Byť se jedná o funkční a elegantní vizualizaci, která zobrazí prvky tak, jak jsou v síti opravdu pospojované, a byť dokonce umožňuje kromě zobrazení základních informací také spoustu dodatečných dat o zařízení (grafy chyb, rychlosti atd., vytížení CPU apod.), stále aplikaci chybí zobrazování požadovaných informací probraných v kapitole 3.1. Z toho vyplývá, že je aplikace nedostačující požadavkům.

Cena balíčku „1000-computer database“ činí $899,95 jednorázově a napořád, což by byla přijatelná cena v případě, že by aplikace splňovala požadovaná kritéria.

3.5.2 NetDepict

Program spolupracuje s aplikací Visio z dílny *Microsoft*. Automatizovaně dokáže současně sbírat data z několika sítí a následně vykalkulovat ideální rozmístění prvků tak, jak jsou mezi sebou propojeny. Zajímavostí na této kalkulaci je, že se snaží sítě rozdělit i co se fyzického rozmístění týče, např. jednotlivé sítě se snaží dávat do jedné hvězdicovité struktury. Je samozřejmě jasné, že tohle fyzické rozmístění bude hodně zkreslené, protože prvky neuchovávají žádnou informaci o poloze.

Po vygenerování vizualizované topologie si uživatel může prvky přesouvat tak, jak sám potřebuje a může vytvářet různé pohledy[4], které si může uložit pro budoucí využití.

Jak píší na jejich oficiální webové stránce (NetDepict, 2012), aplikace také umožňuje „importovat informace o zařízeních z jiných síťových nástrojů, aby bylo možné vykreslit také např. jejich konfigurace atd.“ Bohužel žádné dodatečné informace, návod, či video k tomuto bodu nedodali, tudíž nelze zjistit, které informace aplikace umí zobrazit.

Celkově aplikace vypadá, soudě podle přiloženého uváděcího videa, propracovaně, program je to placený (konkrétně 2 142 Kč za plnou verzi na doživotí), bohužel nenabízí možnost vyzkoušení v podobě trialu nebo dema, tudíž mohou závěry o aplikaci být pouze odhadovány. Z toho, co bylo zjištěno, lze usoudit, že aplikace je nevyhovující, protože nesplňuje více požadavků. Jedním z nich je např. „v místě problému znázornit prvek a vypsat pravděpodobnou chybu“ (viz kapitola 3.1), což aplikace, dle dostupných informací, v takovém případě s vizualizací skončí. Bodů, které nesplňuje, nebo splňuje jen částečně, je však více.

3.5.3 NTM

Akronym *NTM* představuje název aplikace znějící *Network Topology Mapper*. Program je od společnosti *SolarWinds*.

Již na začátku webové stránky (Network Topology Mapper, 2016) se lze dočíst, že cena aplikace začíná na ceně 1220 €, což je dle kurzu dne 18. 11. 2016 33 000 Kč. Tato cena není malá a mohla by být značnou bariérou v případě, že by aplikace splňovala všechny požadavky, nicméně není to nepřekonatelná bariéra.

Jako klíčové vlastnosti aplikace zmiňují:

- automatizované hledání prvků v síti a jejich mapování pomocí protokolů *ICMP, SNMP, WMI, CDP* a další,
- vytváření vícero map z jednoho skenování sítě, což může ušetřit čas, zdroje a šířku pásem,
- exportování map do programu Visio,

[4] V této souvislosti se pojmem „různé pohledy“ myslí různá zobrazení topologie, z nichž každé reprezentuje odlišnou informaci.

- automatické detekování změn v síťové topologii.

Díky možností vyzkoušení 14denní trial verze bylo zjištěno, že aplikace umí zobrazovat základní informace jako je IP adresa, MAC adresa, název zařízení, zařazení do VLAN, virtuální rozhraní, rychlost linek apod., nicméně větší část vyžadovaných informací nedokáže zobrazovat, což je výrazný nedostatek. Dále hledání v síti není, jak je zmíněno v požadavcích, z bodu A do bodu B, ale na základě IP adres podsítí či IP adres tzv. „seed prvků“, které nastíní IP adresy podsítí, ke kterým jsou připojeny.

Aplikace působí jako velmi robustní a připravená pojmout opravdu velké sítě. Pro účely vizualizace jako celku se jeví velmi příznivě, i co se do ceny týče, nicméně pro účely hledané aplikace už, vzhledem k vysoké ceně a zmíněným nedostatkům, příznivá není.

3.6 Výsledky hledání pomocí vyhledávače Google

Výsledků, které nachází vyhledávač *Google* po zadání vymezených klíčových slov, je nespočet. Nicméně spousta nalezených stránek ve skutečnosti se zkoumaným problémem vůbec nesouvisela. Relevantní výsledky budou postupně předloženy, a to jak jejich popis, tak přínos a zhodnocení.

3.6.1 10scape

Aplikace slouží primárně na rychlé vytváření mapy počítačové sítě. Je velmi moderní a uživatelsky přívětivá. Spolupracuje s řešením *SpiceWorks*, které automaticky skenuje síť[5]. Po skenování aplikace vypíše tabulku jednotlivých prvků a umožňuje je přidávat do kreslící plochy. Cena pro verzi „Large“ (250 prvků) činí $125 měsíčně.

Ač je aplikace velmi příjemná a komplexní, slouží pouze na rychlejší kreslení map a nedokáže dynamicky na základě zadaného bodu A a bodu B (resp. jejich IP adres) vykreslit prvky v cestě, tudíž se nedá považovat za použitelnou. Navíc je aplikace závislá na externí diagnostice sítě, což představuje další správu, která navíc nemusí být vůbec triviální.

3.6.2 Cytoscape

Jak uvádí sám web *Cytoscape* (2016): „*Cytoscape* je open source aplikační platforma pro vizualizaci molekulární interakcí sítí a biologických cest. Přestože aplikace byla původně navržena pro biologický výzkum, našla uplatnění ve vizualizaci komplexních sítí. Základní distribuce poskytuje základní množinu funkcí pro analýzu a vizualizaci. Dodatečné funkce jsou přístupné prostřednictvím modulů“.

Rodina *Cytoscape* aplikací poskytuje opravdu zajímavé moduly, které mohou být využity pro různé programovací jazyky, např. *JAVU* či *JavaScript*.

[5] Aplikace ke skenování vyžaduje přihlašovací jména a hesla k různým protokolům zařízení (*SSH, Telnet*, doménové přihlášení atd.

Jako velmi zajímavým modulem pro budoucí vývoj aplikace, v případě nenalezení vhodného současného řešení, se jeví knihovna pro *JavaScript* s názvem *Cytoscape.js*. Nejedná se o kompletní webovou aplikaci, nýbrž slouží jako základní kámen pro webovou aplikaci sloužící na vizualizaci síťové topologie.

Existuje také portál zvaný *Cytoscape App Store*, kde uživatelé sdílejí své vytvořené aplikace, moduly či knihovny, ve kterých využili zmíněnou *Cytoscape* platformu. Výtvorů je zde opravdu mnoho a zcela určitě by se tam dala najít funkcionalita (modul či knihovna), která by byla užitečná v budoucí implementaci aplikace, tudíž se nevylučuje (spíše přímo doporučuje) budoucí prozkoumání tohoto portálu v dalších kapitolách práce.

Základní aplikace *Cytoscape* je určitě zajímavá a pro mnoho oborů přínosná, ale poskytuje vizualizaci komplexních sítí formou interaktivních grafů, kde zobrazuje pouze základní informace, což je pro splnění požadavků aplikace nedostačující. Nicméně díky této analýze se otevřely nové dveře do světa modulů společnosti *Cytoscape* pro případný budoucí návrh a implementaci aplikace.

3.6.3 TNV (The Network Visualizer)

Jak úvod webové stránky sděluje (The Network Visualizer, 2008): „Aplikace je zamýšlená pro analýzu síťového provozu s cílem usnadnit učení o tom, co znamená normální aktivita na dané síti. Vyšetřuje podrobnosti o paketu v otázkách bezpečnosti a ulehčuje troubleshooting."

Základní vizualizace ukazuje vzdálené hosty na levé straně obrazovky a na pravé straně matici lokálních hostů a mezi nimi nakreslenou linku. V lince lze vidět odchozí a příchozí pakety. Zvolením určité buňky v matici lokálních hostů lze zobrazit buď detaily paketů anebo aktivitu portů, patřícím danému uzlu.

Jak už je z popisu aplikace zřejmé, aplikace nebude vyhovující, a to z toho důvodu, že se jedná spíše o vizualizaci provozu na síti než sítě samotné. Nejsou zobrazené prvky ležící mezi vzdáleným a lokálním hostem, což je základní požadavek. Soudě dle toho, co se dá z webových stránek zjistit a vyčíst, dá se aplikace považovat za jakýsi substitut programu *WireShark* v grafickém provedení (resp. vizualizací prvků a toků).

3.6.4 LiveAction

Řešení se vztahuje na vizualizaci cest mezi uzly napříč sítě, včetně L2 informací. Aplikace umožňuje zobrazení veškerých L2 informací, včetně VLAN asociací, trunk rozhraní atd. Díky tomu je možno zjednodušit troubleshooting právě na úrovni L2 a zvýšit bezpečnost sítě nebo např. *QoS*[6].

[6] *QoS* je akronym pro *Quality of Service*, což umožňuje upřednostňování určitého provozu před jiným.

Ačkoliv pro L2 troubleshooting se jedná o velmi kvalitní řešení, včetně automatických detekcí chyb a jejich napravení, pro vyšší vrstvy *ISO/OSI* modelu aplikace nedostačuje, a tudíž se nejeví jako vhodná.

3.7 Shrnutí

Aplikací na vizualizaci síťové topologie existuje relativně velké množství, z něhož každá aplikace se zaměřuje na něco trochu jiného a každá řeší sběr a prezentování dat odlišnými způsoby. Kdyby šlo z každé aplikace extrahovat jen to, co by bylo přínosné pro naše účely, zcela určitě by z toho vzešla aplikace, která by byla použitelná. Bohužel to není proveditelné, tudíž se nabízí jediné řešení, a to navrhnout a implementovat novou vlastní aplikaci, zaměřující se přesně na vymezené požadavky.

Nicméně cíl rešerše, a to prohledání současného stavu aplikací a řešení umožňující vizualizaci sítě na základě daných kritérií a klíčových slov, byl splněn. Díky tomu je nyní jistota, že budoucí aplikace nebude duplikát již nějakého existujícího řešení. Navíc byl objeven portál modulů od firmy *Cytoscape*, které se dají v aplikaci použít.

4 Použité technologie

Kapitola pojednává o technologiích a prostředcích, které sloužily či dopomohly pro zdárné zhotovení aplikace.

4.1 Protokoly pro vzdálený přístup k síťovým zařízením

Aby bylo možné se síťovými prvky vůbec manipulovat a nastavovat je, případně z nich vyčítat potřebné informace, je potřeba se s nimi nějakým způsobem spojit. Toho lze dosáhnout buďto lokálně (místní přístup), např. pomocí konzolového spojení anebo vzdáleně, tj. přes funkční síťovou infrastrukturu (Štěpánek, 2009), k čemuž už se musí využít služby umožňující vzdáleně přistupovat k zařízení.

4.1.1 Telnet

Jednou z možností, jak se spojit se síťovým prvkem, je protokol *Telnet*. Komunikace s prvkem probíhá tak, že se díky tomuto protokolu získá přístup ke konfiguračnímu terminálu, ze kterého už lze vytvářet, číst, upravovat či mazat konfiguraci. Protokol pracuje na síťovém portu 23, na bázi klient-server, kdy klient je na tzv. fyzickém počítači a server na vzdáleném počítači, kam se uživatel chce přihlásit (Správa sítě, 2016).

Dnes bývá často nahrazen protokolem *SSH* z důvodu absence šifrování přenášených dat (včetně hesla při přihlašování), a tedy existencí rizika odposlechu dat během přenosu. Nicméně vzhledem k tomu, že v momentální síťové infrastruktuře na Mendelově univerzitě v Brně většina prvků umožňuje vzdálený přístup pomocí protokolu *Telnet*, je pro vzdálený přístup v této vyvíjené aplikaci využíván právě tento protokol.

4.1.2 SSH

Mezi populární a hojně využívaný nástroj pro vzdálené spojení se řadí protokol *SSH* a jeho nástupce *SSH verze 2*. Na první pohled je velmi podobný protokolu *Telnet*, neboť z něj vychází, protože *SSH* jakožto zabezpečený komunikační protokol vznikl v reakci na špatně zabezpečené (ba přímo nebezpečné) protokoly a příslušné služby typu *Telnet* (dsl.cz, 2017). Protokol opět umožňuje přístup ke konfiguračnímu terminálu síťového zařízení. Pracuje na síťovém portu 22, opět na bázi klient-server.

Hlavním důvodem jeho popularity je šifrování přenášených zpráv a autentizace mezi koncovým zařízením a spravovaným prvkem, také má implementovanou kontrolu integrity dat a případnou bezeztrátovou kompresi. Užitečná také může být autentizace, která využívá vygenerovaných „SSH klíčů“[7], takže odpadá nutnost autentizovat se heslem (Krčmář, 2016).

[7]Na straně klienta (vaše pracovní stanice) musí být přítomen privátní klíč a na straně serveru, k němuž se přihlašujete, klíč veřejný.

4.1.3 SNMP

Mimo výše zmíněné protokoly pro vzdálený přístup k prvku se může také zmínit protokol *SNMP*. Je široce rozšířený a užitečný a slouží k získávání dat či nastavování hodnot na určitém zařízení. Prvkem nemusí být pouze směrovač či přepínač, ale také např. tiskárny, přístupové body atd.

Protokol vyžaduje pro komunikaci dvě strany – správce a agenta. Správce posílá dotazy agentovi a přijímá od něj odpovědi. Na straně správce se používá port 161 a na straně agenta 162.

Každá hodnota v *SNMP* je jednoznačně identifikována pomocí číselného identifikátoru *OID* (Object Identifier), který je součástí stromové struktury. Celá stromová struktura se nachází v *MIB* (Management Information Base) databázi. Právě v ní by se pod konkrétními *OID* dala najít potřebná data určitého síťového zařízení.

Jednou z výhod je podpora *SNMP* ve skriptovacím programovacím jazyku PHP. Díky tomu lze se získanými daty pracovat například i přímo ve webové aplikaci.

4.2 Programovací jazyk JAVA

Pro vývoj backendové části aplikace byl zvolen jazyk JAVA. Důvodů pro zvolení tohoto jazyka je více, nicméně hlavní důvod je ten, že existující univerzitní projekt (resp. jeho část – backend) je psán v tomto programovacím jazyku. Protože jedním z cílů práce je zařadit vyvíjenou aplikaci do tohoto projektu, nelze ani uvažovat nad jiným programovacím jazykem.

Nicméně i kdyby byla možnost jakéhokoliv výběru, nejspíše by i tak zvítězil jazyk JAVA, protože je uživatelsky přívětivý, v porovnání s jinými programovacím jazyky i relativně jednoduchý a má již vybudované velké zázemí a podporu uživatelů. Pokud by se ale měla nějaká alternativa zmínit, stojí za zvážení programovací jazyk Perl.

4.2.1 Obecná charakteristika

Programovací jazyk JAVA je relativně mladý, objektově orientovaný jazyk, který vyvinula firma Sun Microsystems v roce 1995. Dnes je tato firma součástí společnosti Oracle, tudíž je Oracle vlastníkem JAVY a pokračuje v jejím vývoji. JAVA je jeden z nejpoužívanějších a nejpopulárnějších jazyků na světě, zejména díky své robustnosti, podpoře, přenositelnosti, podobnosti s jazykem C++ a využitelnosti. Velmi těžko se hledají oblasti, kde by se JAVA nedala použít jako programovací jazyk (What is JAVA?, 2017). Navíc k tomu všemu se jedná o *open source*[8].

[8] Open source je v doslovným překladu volný zdroj. V tomto případě je tím však myšleno to, že firma v roce 2007 volně zpřístupnila zdrojový kód JAVY komukoliv na světě, a tedy kdokoliv jej může beztrestně využívat a měnit k obrazu svému.

4.2.2 Knihovny

Vzhledem k popularitě tohoto programovacího jazyka existuje velké množství vývojářů, kteří své výtvory (v podobě tzv. knihoven) sdílí napříč Internetem (většinou prostřednictvím repositářů) a kdokoliv jiný si jejich práci může zakomponovat do své vlastní aplikace a naplno ji využívat. Za dobu existence jazyka vzniklo velké množství knihoven, jen zřídkakdy se stane, že by člověk nenašel knihovnu pro určitý problém.

Princip nalezení určité knihovny je velmi jednoduchý – stačí vyhledávat v online repositářích (např. *Maven repository*) pomocí klíčových slov reprezentujících hledanou funkcionalitu. Poté už jen stačí nalezenou knihovnu připojit ke své aplikaci (možností, jak knihovnu připojit, je několik) a poté se ihned mohou využívat její funkcionality, resp. třídy a metody. Většina knihoven má svou dokumentaci, kde je řečeno, co nabízí a jak se s ní má zacházet.

Pro účely aplikace byla využita knihovna od společnosti Apache, nesoucí název *Apache Commons Net*. Jedná se o knihovnu, která implementuje mnoho základních internetových protokolů, jako jsou *FTP*, *SMTP*, atd (Apache Commons Net, 2017). Nicméně byl využit pouze jeden z nabízených protokolů – *Telnet*. Díky tomu jej nebylo potřeba složitě implementovat, což by bylo velmi náročné, protože by bylo potřeba implementovat celou funkcionalitu tak, jak je dáno ve specifikaci protokolu. Takhle stačí knihovnu pouze připojit a vytvořit objekt *TelnetClient*, který se již postará o správné fungování protokolu, aniž by bylo potřeba vědět, jak funguje, což je výhoda objektového přístupu, které se říká *zapouzdření*.

4.3 Knihovna jQuery

Vzhledem k tomu, že vyvíjená aplikace bude mít i část, která poslouží pro interakci s uživatelem prostřednictvím webové stránky v prohlížeči, bude nezbytné využít prostředky pro tvorbu webových stránek. Jedná se o značkovací jazyk *HTML*, zajišťující strukturu a vyobrazení webové stránky, jazyk *CSS* (kolekce stylů pro grafickou úpravu webových stránek) a programovací jazyk *JavaScript*, zajišťující funkční stránku webové aplikace. Všechny tyto jazyky jsou „profláknuté“ a i laik mnohdy tuší, o co se jedná. Nicméně pro implementaci bude potřeba využít nástavby *JavaScriptu* – Knihovy *jQuery*, která *JavaScript* obohatí o mnoho užitečných vlastností.

JQuery je nejrozšířenější *JavaScriptová* knihovna s jednoduchou syntaxí (Čápka, 2017). Stačí ji opět pouze připojit k aplikaci a poté je možné využívat všechny její nabízené funkce, zde je jejich částečný výčet: (JQuery, 2017):

- výběr DOM elementů pomocí otevřeného cross-browser selektorového enginu Sizzle, odnože projektu *jQuery*,
- funkce pro procházení a změnu *DOM* (Document Object Model – objektový model dokumentu),
- události,

- manipulace s *CSS*,
- efekty a animace,
- *AJAX*,
- rozšiřitelnost,
- utility – např. informace o prohlížeči nebo funkce *each*.

Pro účely vyvíjené aplikace je nejdůležitější a nejpoužívanější nabízená funkce *AJAX* (Asynchronous JavaScript and XML). *AJAX* je technologie pro přístup k webovým serverům z webové stránky. Dále umožňuje dynamicky měnit obsah, aniž by se stránka musela celá znovu načítat (AJAX Introduction, 2017). Dělá to tak, že na základě výsledku asynchronní komunikace se serverem aktualizuje obsah stránky podle toho, jak komunikace se serverem dopadla. Jakmile komunikace skončí, *AJAX* učiní zpětné volání, tzv. „callback“, které případně může změnit obsah stránky. To vše provádí na pozadí, tudíž uživatel nemusí čekat, než se vrátí data s odpovědí.

Protože *AJAX* je již součástí knihovny *jQuery*, nemusí vývojář řešit, jak se celá taková asynchronní komunikace provádí. Postačí, když v kódu zavolá metodu `ajax()`(JQuery ajax() Method, 2017) a předhodí ji potřebné parametry. Nejdůležitějšími z nich jsou `url, success` a `error`. Url slouží k přesné specifikaci umístění serveru, resp. umístění zdrojů informací. Parametry success a error určují, co se stane v případě úspěšného (success) či neúspěšného (error) obdržení dat, většinou prostřednictvím definované funkce.

4.4 Neo4J databáze

4.4.1 Úvod

V dnešní době existuje mnoho různých způsobů jak ukládat data. Asi nejznámějším a nejpoužívanějším způsobem jsou relační databáze, zejména z toho důvodu, že jsou již velmi rozšířené u malých společností až po ty nadnárodní a jsou také jednoduché na použití díky dotazovacímu jazyku *SQL*, který se snaží napodobit lidskou řeč.

Další možností je využít grafové databáze, které nabízí některé funkcionality a vlastnosti, jež by se v relačních databázích velmi obtížně implementovaly. Např. konkrétně jde o funkcionalitu, ve které grafové databáze vynikají – jednoduché nalezení cest z bodu A do bodu B včetně všech prvků mezi nimi. Toho je dosaženo pomocí grafových algoritmů pro nalezení nejkratší cesty. Navíc samotná reprezentace dat v grafové databázi se velmi podobá reprezentaci prvků v počítačové síti.

V relačních databázích existují entity (resp. tabulky) se svými atributy[9] a vazbami mezi nimi. V grafových databázích se jedná o tzv. uzly (anglicky Nodes) se svými vlastnostmi (anglicky Properties) a vztahy mezi nimi (anglicky Relati-

[9]Atributy identifikují určité vlastnosti jedné konkrétní reprezentace entity.

onships), které jednotlivé uzly propojují[10]. Uzly mohou mít nepovinně jedno nebo i více označení (anglicky Labels), která je seskupují dohromady (What is a Graph Database, 2017).

4.4.2 Dotazovací jazyk *Cypher*

Všechny databáze mají nějaký dotazovací jazyk, který slouží pro tvorbu, čtení, upravování a mazání jak struktury databáze, tak samotných dat. V grafové databázi *Neo4J* je k dispozici dotazovací jazyk, nesoucí název *Cypher*. Ti, kteří mají zkušenosti s dotazovacím jazykem *SQL*, mohou postřehnout velmi podobnou syntaxi i v *Cypheru*. S jazykem se pracuje tak, že se zadá, **co** má najít, **nikoliv jak** to má najít.

Syntaxe pro vytvoření dat je velmi jednoduchá a intuitivní (Intro to Cypher, 2017), může mít např. takovou podobu:

```
CREATE (S1:Device {hostname: 'switch1', username: 'admin', password:
    'cisco'})
```

Tímto příkazem jsme vytvořili uzel S1 (S1 reprezentuje proměnnou), který má označení „Device“ a vlastnosti hostname s hodnotou „switch1“, username s hodnotou „admin“ a password s hodnotou „cisco“.

Syntaxe pro hledání dat může vypadat například takto:

```
MATCH (sw:Device) WHERE sw.hostname = "switch" RETURN sw
```

Klíčovým slovem „MATCH“ specifikujeme, že se jedná o hledání, resp. porovnávání v databázi. V tomto konkrétním případě hledání omezíme (uvedením klíčovým slovem „WHERE“) pouze na prvky, které mají vlastnost hostname s hodnotou „switch1“.

Protože síla grafové databáze tkví zejména ve vztazích mezi jednotlivými uzly, je nutné tyto vztahy umět vytvořit. Konkrétní vztah může být vytvořen například takto:

```
(S1)-[:CONNECT {speed: 100}]->(R1)
```

Tímto je vytvořen vztah mezi prvkem S1 a prvkem R1, který říká, že S1 spojuje („CONNECT“) R1, navíc s vlastností rychlost („speed“) s hodnotou 100 (což může označovat rychlost např. 100 Mb/s). Vztah má orientaci směrem z S1 do R1.

Samozřejmě existují dotazy na další práci s databází i daty, které mohou být mnohdy velmi komplexní a pro důkladné pochopení problematiky by si to zasloužilo samostatnou práci. V této sekci šlo zejména o to, ukázat podobnost s dotazovacím jazykem *SQL* a také nastínit základní operace s grafovou databází.

[10]Každý vztah má daný směr a může obsahovat vlastnosti stejně tak, jako obsahují uzly.

4.5 In-memory databáze

Grafová databáze se nehodí na ukládání některých typů dat. Jedná se například o data typu výčet apod. K tomuto účelu se spíše hodí databáze typu *SQL*. Nicméně pokud takových dat není mnoho, mnohdy se nevyplatí dedikovat prostředky a vytvářet samostatný *SQL server*, který by spravoval databázi.

Zajímavým kompromisem je tzv. *in-memory* databáze[11], spolupracující s jazykem *SQL*. Je již součástí aplikace a ukládá se přímo do hlavní paměti (většinou RAM), kde je uložená i samotná aplikace, což umožňuje velmi rychlý přístup k datům (Rouse, 2012). Databázi mnohdy není ani potřeba explicitně definovat a vytvářet její strukturu, neboť existuje spousta nástrojů, které se o to postarají za vývojáře.

Samozřejmě je zde možnost spojit *in-memory* databázi s databází grafovou. To bývá většinou realizováno způsobem, kdy uzel grafové databáze obsahuje atribut, ve kterém je uložena hodnota ID (jedinečného identifikátoru), určující konkrétní objekt (resp. záznam) v *in-memory* databázi.

[11]Někdy se také používá pojem *Embedded* databáze – zabudovaná databáze. Většinou se jedná o synonymum.

5 Návrh aplikace

Tato kapitola pojednává o návrhu vyvíjené aplikace. Jak bylo zmíněno již dříve, aplikace bude sloužit pro vizualizaci cesty v počítačové síti od bodu A k bodu B. Přesné vlastnosti a specifikace aplikace jsou popsány v kapitole 5.2. Vzhledem k tomu, že je potřeba tuto funkcionalitu zakomponovat do již existujícího projektu, je postupně rozebráno i to, co bude potřeba dodat, odebrat, či pozměnit v tomto projektu. Dále bude popsáno, jak je navrhnuta samotná část, řešící vizualizaci včetně návrhu algoritmů pro správný běh.

5.1 Momentální stav existujícího projektu

Existující projekt resp. aplikace ve vývoji je vyvíjená již od roku 2016 Ing. Janem Kolomazníkem, Ph.D. a Ing. Jiřím Lýskem, Ph.D. Jejím úkolem je převzít správu síťové infrastruktury, která momentálně běží v informačním systému univerzity na zcela jiné aplikací. Vyvíjená aplikace je rozdělena na dvě části – část frontend a část backend.

5.1.1 Frontend

Část frontend se stará o vše, co je spojeno s uživatelem. Jde tedy zejména o vzájemnou interakci, jako je vkládání určitých údajů pro další práci, zadávání dotazů a úkolů atd. a poté zobrazení výsledků na základě požadavku od uživatele. To vše je realizováno prostřednictvím webové aplikace.

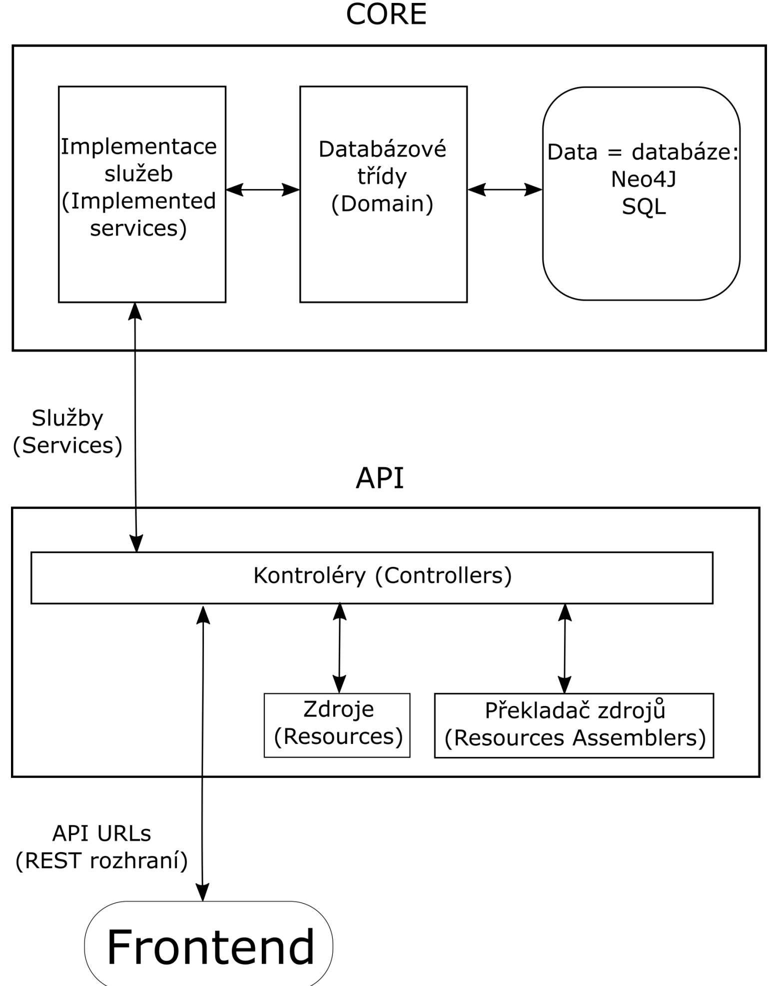
Jakmile je zadán požadavek od uživatele, včetně případných údajů, je zavolána příslušná služba v části backend, která požadavek zpracuje a dodá výsledky frontendu, který je zobrazí zpět uživateli. Vzhledem k tomu, že je existující frontend napsán v nástavbě *AngularJS*, pro účely této práce je vyvíjený frontend psán samostatně, nezávisle na tomto existujícím řešení, zejména pro demonstraci funkčnosti.

5.1.2 Backend

Backend je již část, kde se provádějí největší změny vzhledem k vyvíjené aplikaci. Slouží pro práci s daty – jejich vytváření, čtení, aktualizování a mazání. Má dvě části – *CORE* (jádro) a *API* (Application Programming Interface). *CORE* pracuje s daty (operace *CRUD*[12]). *API* zpřístupňuje rozhraní pro komunikaci s částí frontend.

Jak je vidět na obrázku 1, část *CORE* se skládá ze tří dílů, kdy každý má jiný účel. Díl „Data“ reprezentuje databázi, kde jsou ukládána veškerá aplikační data (buďto za pomocí grafové databáze *Neo4J* nebo za pomoci *in-memory SQL databáze*).

[12] *CRUD* je akronym, kdy písmeno C stojí za slovem Create (vytvořit), R za Read (číst), U za Update (Aktualizovat) a D za Delete (smazat). Značí tedy, že s určitými daty lze dělat jakékoliv možné operace.



Obrázek 1: Propojení backendových částí API a CORE

Další díl „Databázové třídy“ má za úkol přistupovat k datům a definovat databázovou strukturu. Obsahuje jak třídy reprezentující konkrétní objekty (v případě *in-memory databáze* tabulky) v databázi, tak repositářové třídy[13].

Třetí, poslední díl „Implementace služeb“, obsahuje třídy implementující služby (Services), resp. jejich metody, neboť služby představují třídy typu *Interface*[14]. Celá část *CORE* pak s částí *API* komunikuje právě prostřednictvím zmíněných služeb (Services).

Část *API* se skládá opět ze tří dílů. Díly „Zdroje“ a „Překladač zdrojů“ slouží ke správnému překladu objektů databázových tříd na objekty použitelné pro část frontend (ve formátu *JSON*). Poslední díl „Kontroléry“ poskytuje jak rozhraní pro

[13]Repositářové třídy provádějí operace nad uloženými daty. Kromě základních *CRUD* operací si vývojář může vytvořit jakoukoliv svou.

[14]*Interface* je v JAVĚ typ třídy, který pouze definuje metody, které musejí být v jiných třídách implementovány. Dělá se to zejména z důvodu větší přehlednosti a škálovatelnosti.

část frontend, tak i využívá prostředky *CORE* prostřednictvím služeb.

Rozhraní pro frontend jsou realizována pomocí *URL*, které mají stejnou reprezentaci jako webové stránky. Podoba URL tedy může být např. taková:

```
localhost:8080/api/devices
```

„localhost“ značí, že backend běží na lokálním serveru (resp. stroji), číslo za dvojtečkou je síťový port, na kterém backend „naslouchá“. Text za lomítkem pak už specifikuje dotaz na konkrétní služby. Ty jsou volány prostřednictvím kontrolérů, které je zpřístupňují. Celá komunikace je založena na principu *REST*[15].

Principy (databázové třídy, repositáře, služby, kontroléry apod.), popsané v této kapitole, vychází z konvence, jež je specifikována ve *Spring Frameworku*. Konkrétně se pak jedná o rodinu *Spring Data*, resp. její část *Spring Data JPA* (Spring Data JPA – Reference Documentation, 2016).

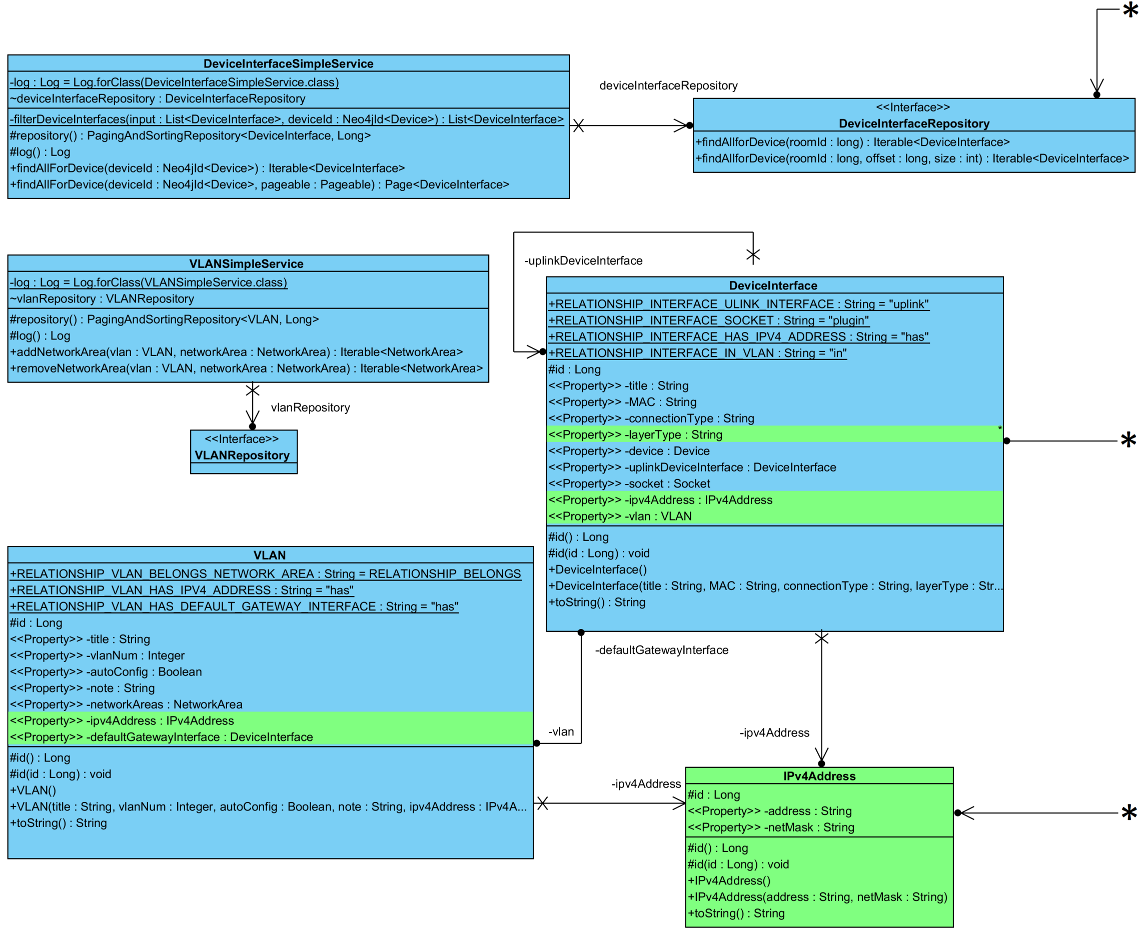
5.1.3 Upravený návrh backendu – část CORE

Vzhledem k tomu, že při vyvíjení existujícího projektu vývojáři nemuseli mít přesnou představu o tom, jaké všechny funkcionality bude výsledná aplikace nabízet, je zřejmé, že spousta věcí se musí pozměnit či dodat v průběhu. Tohle nemine ani vyvíjenou aplikaci pro vizualizaci sítě. Nejvíce změn se uskutečňuje v backendové části *CORE*. Je totiž potřeba v databázi uchovávat o dost více údajů, než bylo uchováváno doposud.

Přidané atributy (tzv. *property*) v databázových třídách, potažmo i celé třídy, jsou znázorněny na dílčích obrázcích 2 a 3. Lze na nich vidět třídy tří typů – **databázové třídy** (poznají se tak, že nemají žádná přídavná klíčová slova), **třídy repositářové** (poznají se podle přidaného klíčového slova „Repository“ za názvem) a **třídy implementující služby** (poznají se podle přidaných klíčových slov „SimpleService“ za názvem), viz kapitola 5.1.2. Zeleně vyznačené části reprezentují přidané vlastnosti, červeně vyznačené pak vlastnosti odebrané. Modře obarvené části reprezentují stav, který byl již součástí existujícího projektu. Vzhledem k tomu, že vyvíjená aplikace pro vizualizaci nevyužívá všechna uložená data v databázi, není zde uveden celý návrh databázových tříd, nýbrž pouze ty třídy, které jsou používané.

Na obrázku 2 je znázorněno 7 tříd. Třída „DeviceInterface“ (a také další třídy „DeviceInterfaceSimpleService“ a „DeviceInterfaceRepository“) pracuje s informacemi o rozhraní určitého zařízení. Vlastnost „layerType“ určuje, zdali se jedná o rozhraní pracující na *ISO/OSI* (Kurose, 2014, str. 60) vrstvě úrovně 2 (L2 rozhraní) či úrovně 3 (L3 rozhraní). Každé rozhraní nově může uchovávat také IP adresu verze 4 (pokud rozhraní pracuje na úrovni L3, což specifikuje zmíněná, nově přidaná vlastnost), potažmo do jaké VLAN patří (opět volitelná vlastnost na základě „layerType“). Třída „VLAN“ nově uchovává IP adresu, která reprezentuje adresu sítě,

[15]„Rozhraní REST je architektura rozhraní, navržená pro distribuované prostředí. Je použitelné pro jednotný a snadný přístup ke zdrojům (Resources). Všechny zdroje mají vlastní identifikátor URI“ (Malý, 2009).

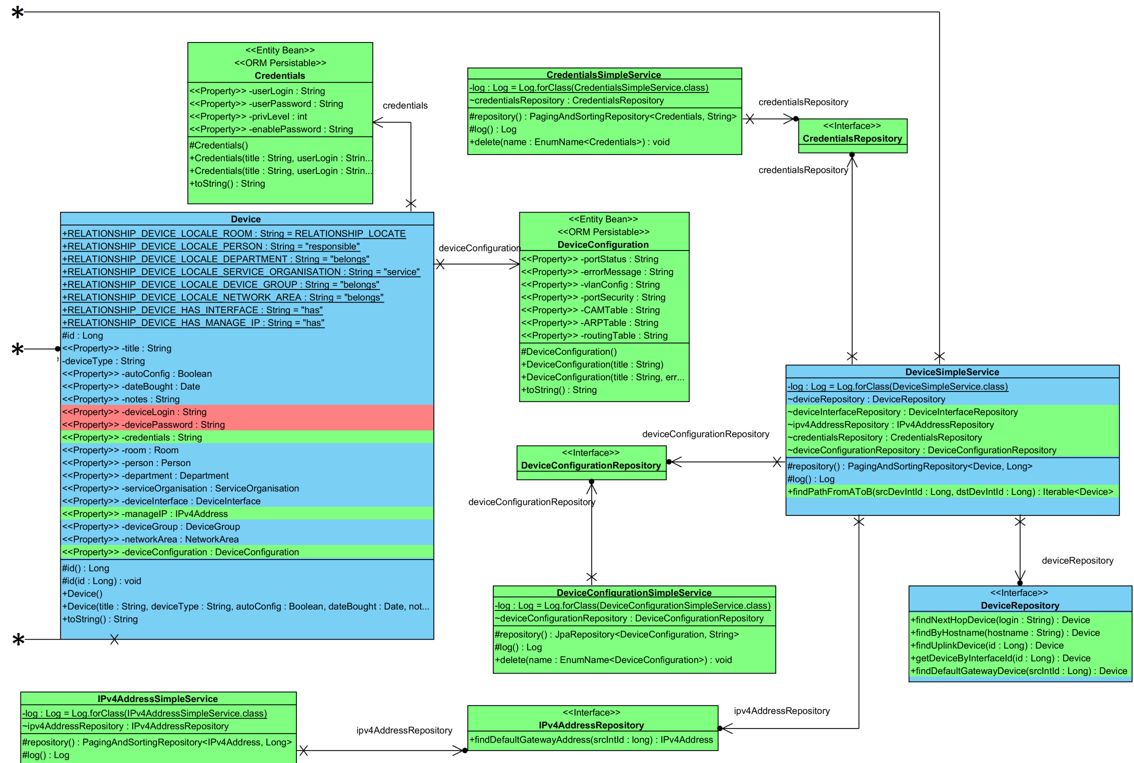


Obrázek 2: UML návrh změn v části CORE – 1. část

a také výchozí bránu, což je provedeno prostřednictvím ukazatele na určité rozhraní, které slouží jako výchozí brána pro tuto VLAN.

Vzhledem k tomu, že se ve vícero třídách ukládá IP adresa, byla vytvořena samostatná třída, ukládající jak IP adresu, tak masku adresy, nazvána „IPv4Address“. Jak je z názvu této třídy patrné, ukládá IP adresu verze 4. Jsou tedy zde do budoucna otevřené dveře k obohacení aplikace na ukládání IP adres verze 6.

Na obrázku 3 je zobrazeno 11 tříd. Je zde znázorněna jedna z nejdůležitějších tříd – třída „Device“. Reprezentuje ukládané zařízení, což je *alfa a omega* celé vizualizace. Dříve byly ukládány přihlašovací údaje k zařízení v podobě atributů. To bylo zrušeno a nově byla vytvořena třída „Credentials“, ukládající potřebné informace o přihlašovacích údajích (údaje jsou ukládány v *in-memory* databázi). Cisco zařízení mají většinou nastaveny přihlašovací údaje ke vzdálenému přístupu (k tomu slouží „userLogin“ a „userPassword“) a na základě úrovně oprávnění („privLevel“)



Obrázek 3: UML návrh změn v části CORE – 2. část

je pak volitelně ukládáno heslo do privilegovaného módu („enablePassword“). Dále je přidán atribut „manageIP“, odkazující na IP adresu, pomocí které je možno se na zařízení vzdáleně připojit.

Aplikace by měla umět také uživateli zobrazit základní konfigurační údaje u každého prvku, jak je uvedeno v kapitole 3.1. To je vyřešeno pomocí nové databázové třídy „DeviceConfiguration“, která nedělá nic jiného, než že uloží u každého Cisco prvku aktuální konfiguraci určitých položek (např. tabulky CAM či ARP). Existuje také atribut „errorMessage“, který slouží ke specifikaci případné chyby, která se vyskytla při pokusu o nalezení cesty. Opět je tohle ukládáno v *in-memory databázi*, neboť konfigurace jednotlivých zařízení se mohou v reálném čase měnit, a proto je rychlejší a jednodušší je udržovat v tomto typu databáze, jak bylo vysvětleno dříve.

Třída „DeviceRepository“ obdržela pět nových metod pracujících v databázi se zařízeními. Všechny budou využívány při hledání cesty. Nejdůležitější přidaná metoda se nachází ve třídě „DeviceSimpleService“. Nazývá se „findPathFromAtoB“ a jejími parametry jsou zdrojové a cílové rozhraní[16]. Tato metoda je implementací

[16] Tím, že se nově u každého rozhraní typu L3 uchovává IP adresa, je vhodné brát jako počáteční a koncový bod právě rozhraní, neboť ty nám jasně určí, jaká bude zdrojová a cílová IP adresa. Zároveň také indikují, jaký je zdrojový a cílový prvek, protože každé rozhraní musí být součástí právě jednoho konkrétního prvku.

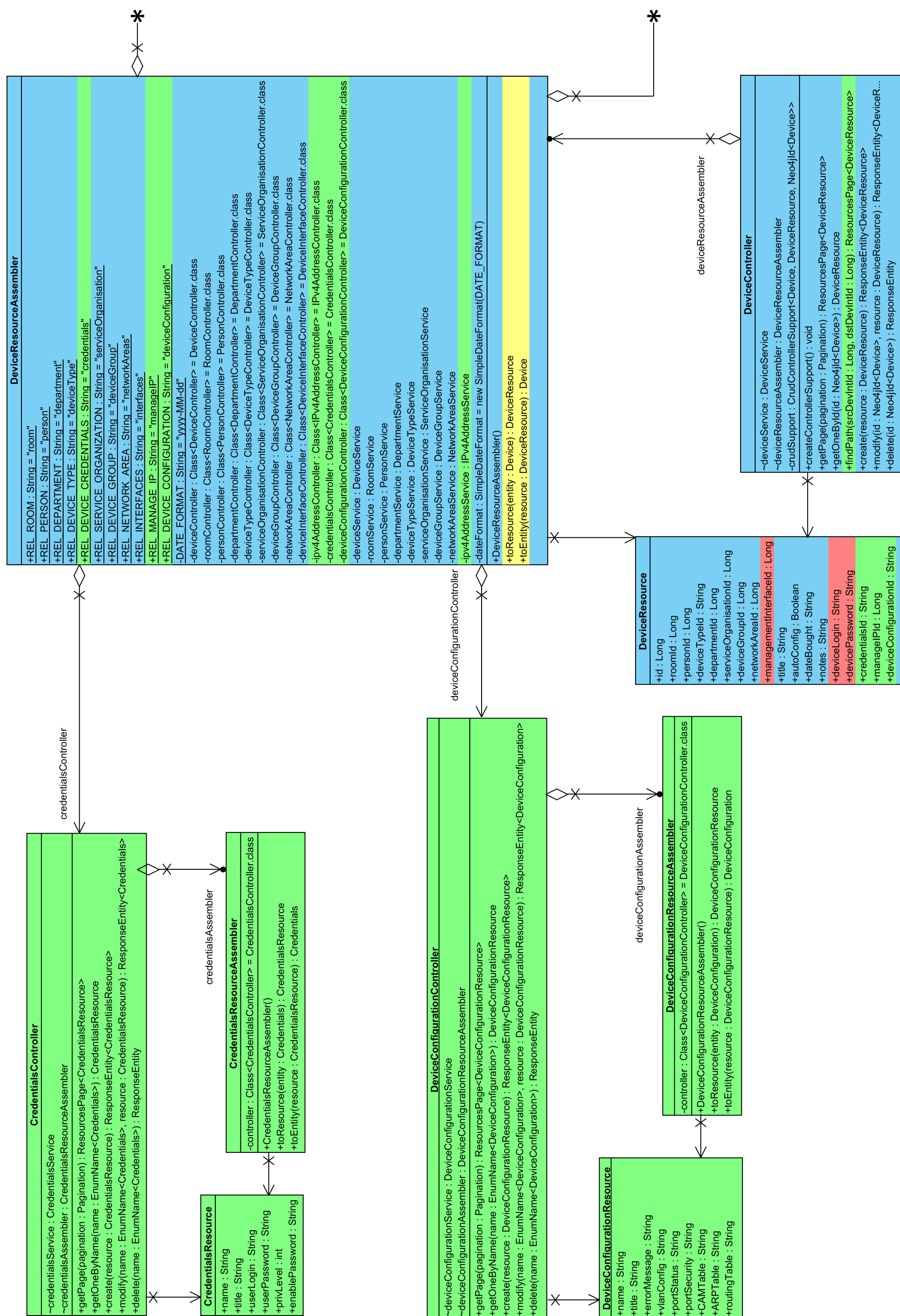
stejnojmenné metody, nacházející se v příslušné službě. Slouží ke spuštění algoritmu pro nalezení cesty z bodu A (resp. ze zdrojového rozhraní) do bodu B (do cílového rozhraní).

Nerozdělený UML návrh změn lze najít v přílohách (složka „Kompletní UML diagramy“).

5.1.4 Upravený návrh backendu – část API

Navržené změny v backendové části *API* jsou svým způsobem velmi podobné změnám v části *CORE*. Má to své opodstatnění, protože část *API* vychází z části *CORE*. V podstatě *API* jen zrcadlí vše, co je v *CORE* části. Zatímco v části *CORE* byly navrženy třídy hlavně pro definování struktury pro ukládání dat a práci s nimi, v části *API* je definováno, jak má vypadat výstup z části *CORE* pro uživatele resp. frontend. Vše je navrženo tak, aby byl výsledný výstup typu *JSON*.

V části *API*, jak již bylo zmíněno v kapitole 5.1.2, se nachází tři typy tříd – kontroléry, zdroje a překladače zdrojů. Návrh ilustrují opět dva obrázky – obrázek 4 a obrázek 5 (nerozdělený UML návrh lze opět najít v příloze). Dohromady tvoří celkový návrh *API*, nicméně aby jednotlivé položky na obrázcích byly čitelné, byl návrh rozdělen. V obrázcích jsou přítomny 4 barvy. Modrá barva značí to, co už je navrženo v existujícím projektu, zelená barva pak přidané změny, červená barva odebírané věci a žlutá barva metody, které již byly součástí existujícího projektu, ale bylo nutné je pozměnit implementačně tak, aby fungovaly správně s novými změnami.



Obrázek 4: UML návrh změn v části API – 1. část

Obrázek 5: UML návrh změn v části API – 2. část

V první části návrhu (obrázek 4) jsou vidět třídy, starající se o zařízení. Třída „DeviceController“ má přidanou metodu *findPath()* s parametry ID zdrojového a cílového rozhraní. Metoda se dá chápat jako spoušť pro algoritmus, vyhledávající cestu v síti. Volá ji frontend a očekává, že mu metoda vrátí *JSON* výstup, ve kterém budou všechna zařízení, která se nachází v cestě. Ve třídě „DeviceResourceAssembler“ přibyly nutné atributy (agregované kontroléry, služby či atributy popisující vztah) pro správný překlad zařízení na typ *JSON*. Třída „DeviceResource“ pouze reflektuje změny provedené v části *CORE*. Všechny atributy, které by měly mít vazbu na jinou třídu, jsou ukládány datovým typem *Long* (pro grafovou databázi) či *String* (pro *in-memory databázi*), protože se jedná pouze o odkazování na ID daného objektu v databázi.

Nově přibyly třídy pro operaci s přihlašovacími údaji („Credentials“) a pro konfiguraci zařízení „DeviceConfiguration“. Opět třídy reflektují návrh v části *CORE* a navrhují atributy a metody, které jsou nutné pro správný běh.

V druhé částí návrhu (obrázek 5) jsou znázorněny třídy pro práci s rozhraními, VLAN a IP adresami. Ve třídě „DeviceResourceAssembler“ přibyly kromě nově agregovaných kontrolorů či služeb, také dva atributy, popisující, s čím je třída ve vztahu a jak by se dalo vztah pojmenovat. Pokud se pečlivě studoval návrh části *CORE*, lze namítnout, že zde chybí návrh pro vztah rozhraní-VLAN. Vzhledem k tomu, že rozhraní může mít *1 až N* přiřazených VLAN, ve kterých pracuje, nelze jednoduše přidat atribut, který by tohle ukládal[17]. Vztah 1:N se řeší interně ve třídě „DeviceInterfaceResourceAssembler“, která sestaví *JSON* v takové podobě, kde atribut pro VLAN bude rozvětvený další *JSON* s obsaženými VLAN. Podobně to řeší např. třída pro zařízení, ve které existuje vztah zařízení-rozhraní, který je také 1:N.

Ve třídě „VLANResourceAssembler“ a „VLANResource“ byly zakomponovány změny pro ukládanou IP adresu a výchozí bránu sítě dané VLAN. Také přibyly třídy pro práci s IP adresami, opět jsou zde obsaženy všechny atributy a metody, které jsou nezbytné pro správnou funkčnost.

Následující podkapitoly se budou už týkat pouze aplikační části pro hledání cesty a následnou vizualizaci.

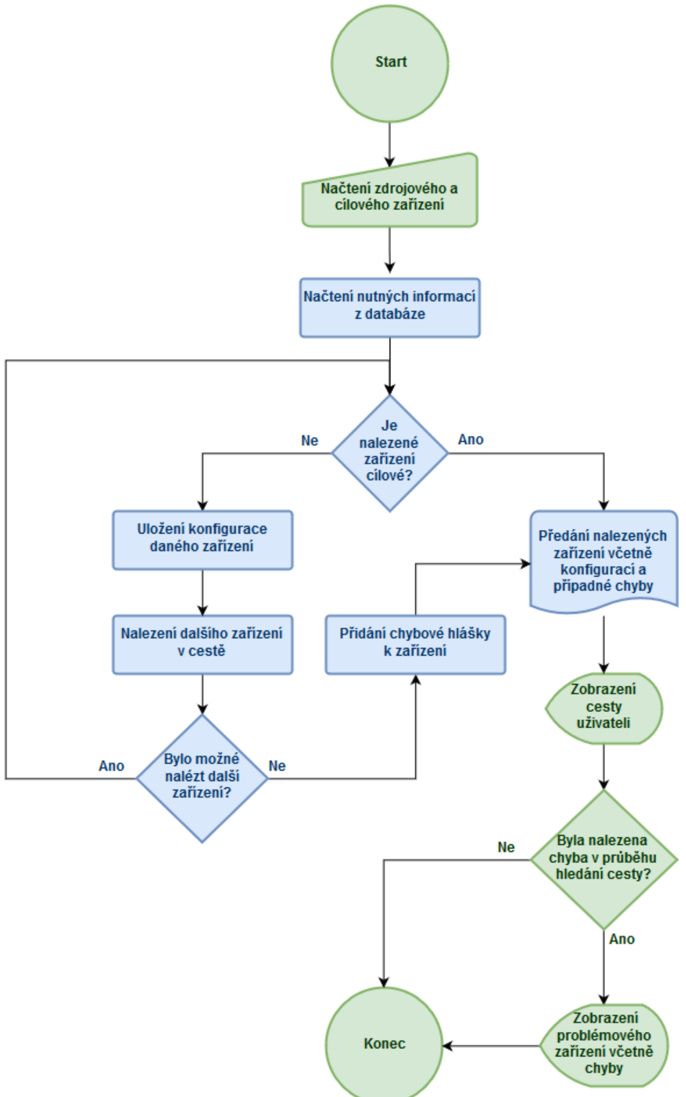
5.2 Specifikace vyvíjené aplikační části

Stručná specifikace aplikace je již uvedena v kapitole 3.1. To všechno samozřejmě platí i nadále, nyní je však prostor pro podrobnější návrh a specifikaci. Veškeré další podkapitoly popisují podrobný návrh vyvíjené aplikace, která se stará již pouze o vyhledání cesty z bodu A do bodu B a následného vykreslení nalezených zařízení v cestě.

[17] Pokud se člověk pozorně podívá na návrh, v každé třídě typu Zdroj (Resource) jsou pouze atributy jednoduchého datového typu. Nikdy zde nenalezneme např. datový typ pole.

5.3 Funkční stránka vyvíjené aplikační části

Funkční stránku vyvíjené aplikace pomůže vyjasnit vývojový diagram na obrázku 6, který znázorňuje, jak aplikace postupuje v nalezení cesty. Jedná se o zjednodušení a je zřejmé, že mohou nastat stavy či činnosti, které nejsou v diagramu obsaženy, nicméně to ani není účelem. Zelená barva u jednotlivých operací v diagramu značí, že je úkon prováděn v části frontend, modrá v části backend.



Obrázek 6: Vývojový diagram vyvíjené aplikace

Počáteční (zdrojový) bod A a koncový (cílový) bod B v počítačové síti určí uživatel v části frontend. Aplikace na základě těchto informací vyhledá v grafové databázi konkrétní prvky a další nutné informace, jako je zdrojová a cílová IP adresa.

Poté již pracuje samotný algoritmus pro nalezení cesty (přesný návrh algoritmu viz kapitola 5.4.3). Ten postupně vyhledává všechny prvky v cestě k cílovému bodu B. Ke každému prvku se vždy spojí a provede následující úkony:

- pokud není prvek cílový, pokusí se najít další prvek v cestě (na základě konfigurace současného prvku zjistí, kudy by měla data dále téct k cílové IP adrese),
- uloží důležité informace o zařízení, které zjistí z příkazového řádku (zjišťované informace závisí na typu zařízení),
- v případě, že nešlo určit další prvek v cestě, zjistí příčinu a uloží ji ke konfiguraci pro další zpracování.

Jakmile algoritmus pro hledání cesty ukončí svou činnost a vrátí nalezená data, mohou nastat dvě varianty:

1. šlo najít kompletní cestu z bodu A k bodu B bez jediné chyby,
2. v cestě se objevila chyba, zabraňující najít další prvek (chyb může nastat nespočet – nemožnost spojit se s prvkem, špatná konfigurace atd.).

Ať už to dopadne jakkoliv, získaná data jsou předána části frontend, který vizualizuje ty prvky, které byly nalezeny. K nim dodá sesbírané informace, které si může uživatel „rozkliknout“. Pokud nebyla nalezena kompletní cesta, je poslední prvek zvýrazněn jako chybový a zobrazí se uživateli chybová hláška, která určí, jaký byl důvod neúspěchu při pokusu o nalezení dalšího prvku.

5.4 Backend

Backendová část tu již byla představena v kapitole 5.1.2. Byl zde také předložen návrh změn v současném stavu existující aplikace (viz kapitoly 5.1.3 a 5.1.4). Tím byla připravena „půda“ pro část aplikace, která se stará o nalezení cesty (o vizualizaci nalezené cesty se stará frontend, jež je popsán v kapitole 5.5). Celé nalezení cesty probíhá v části *CORE*, neboť je potřeba mít přístup k datům uložených v databázích.

5.4.1 Objektový návrh

Objektový návrh je znázorněn na obrázku 7. Tento návrh pokrývá návrh celé části, která se stará o nalezení cesty. Jsou zde zobrazeny všechny třídy, které jsou potřeba k celkovému běhu:

- třídy pro spojení se s prvkem,
- třídy pro rozebírání (tzv. „parsování“) dat z příkazové řádky zařízení,
- třídy poskytující určité dílčí informace ze zařízení (např. informace ze směrovací tabulky, z ARP tabulky atd.),
- třída řídící celý proces.

Třída, řídící celý proces, je „PathFinder". V ní je realizován samotný algoritmus pro nalezení cesty (viz kapitola 5.4.3). Třída reprezentuje celé nalezení cesty „navenek" ostatním třídám, zejména pak třídám typu Service (služba), které mohou veřejné metody této třídy používat. Kromě konstruktoru má pouze jednu veřejnou metodu *findPathByDeviceInterfaces()*, která spouští celý proces hledání cesty a vrátí objekt, který reprezentuje všechna zařízení v cestě. Jak název metody napovídá, je hledání cesty realizováno pomocí zdrojového a cílového rozhraní zařízení. Je zde tedy prostor pro budoucí rozšíření o další možnosti zadávání zdrojového a cílového bodu (např. podle MAC adresy). Zatímco v této metodě je obstaráno vše důležité pro začátek hledání (nastavení zdrojové a cílové IP adresy, nalezení zdrojového a cílového zařízení v databází atp.), v privátní metodě *findPath()* je realizován zbytek algoritmu pro hledání cesty. Metoda „findPath()" je univerzální, nehledě na to, jakým způsobem je dodán zdrojový a cílový bod, tudíž je znovupoužitelná i pro zmíněné možné rozšíření. Ostatní metody této třídy jsou privátní a slouží pro potřeby obou zmíněných metod. Jejich názvy, vstupní atributy a návratové hodnoty by měly dostatečně objasňovat funkcionalitu a použitelnost.

Třída využívá některé repositářové třídy (viz kap. 5.1.2) a také třídu „Device", jejíž objekty slouží jako návratová hodnota hlavní metody pro nalezení. Z atributů je vhodné zmínit „deviceCommunicator". Ten reprezentuje objekt třídy „DeviceCommunicator", který zpřístupňuje komunikaci s aktuálním zařízením v cestě. Je tedy zřejmé, že se v průběhu mění v závislosti, jak se postupně nalézají zařízení.

Třída „DeviceCommunicator" (dále jen komunikátor) zaštiťuje komunikaci se zařízením. Zpřístupňuje třídě „PathFinder" spoustu nezbytných metod, např. pro připojení se k prvku (*connectToDevice()*), poslání příkazu zařízení (*sendCommand()*), zjištění MAC adresy na základě IP adresy (*getMACByIP()*) atd. Jedná se o abstraktní třídu, tudíž je nezbytné, aby měla potomky. Třída „L2Communicator" tohle zajišťuje – implementuje abstraktní metody a navíc přináší metody nové. Objekt této třídy reprezentuje komunikátor, pracující s L2 zařízením (tzn. přepínačem). Třída je ještě navíc předkem třídy „L3Communicator", která mění některé metody (tak, aby reflektovaly funkcionalitu na L3 zařízení) a opět dodává metody nové.

Metoda *connectToDevice()* se zkouší připojit k zařízení a vrací objekt definovaný třídou „DeviceCommunicationResult". Ta má atributy pro uložení případné chyby, která nastala při pokusu o připojení, a pro zjištění výsledku připojení. Její potomci pak definují konkrétní typ chyby. Zde je možnost rozšíření o další, konkrétnější potomky, nicméně momentální stav dvou potomků je naprosto dostačující, protože pokrývá jak úspěšnou (třída „NoErrorResult"), tak neúspěšnou („ConnectionNotCreatedResult") komunikaci.

Komunikátor agreguje třídu „DataParser" a její objekt si ukládá do atributu „dataParser". Tato třída je zodpovědná za správné rozebírání (tzv. parsování) dat. Problém je totiž takový, že zařízení na základě určitého příkazu vydává odpovědi v textové formě (tzv. surová data), které jsou sice formátovaná do tabulky, nicméně taková reprezentace je pro další práci v kódu velmi složitá. Tento problém řeší zmíněná třída, která surová data (např. textová data, reprezentující směrovací ta-

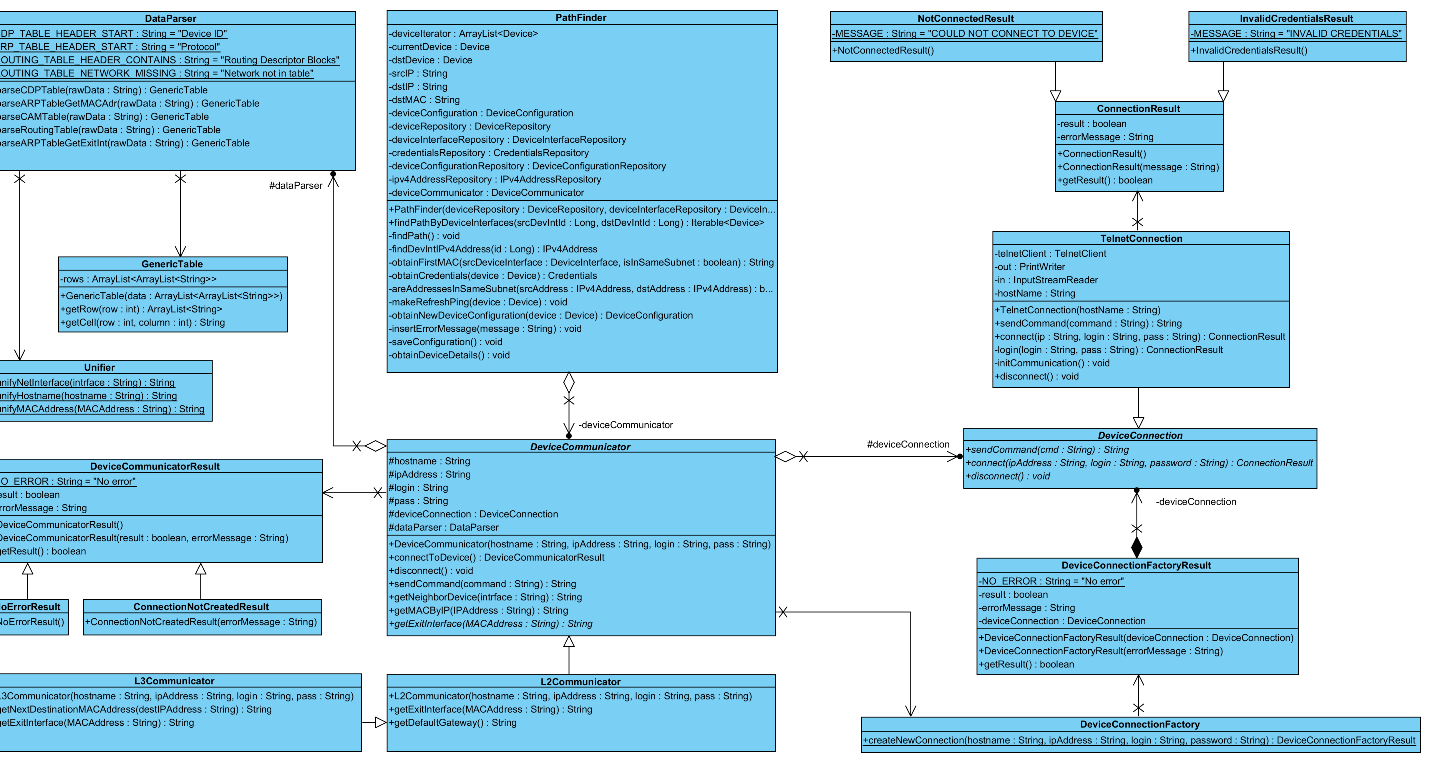
bulku získanou dotazem `show ip route`) „vyparsuje" do přehledné tabulky, resp. do objektu, popsaného ve třídě „GenericTable", který má v sobě atribut, ukládající textovou tabulku do podoby maticové tabulky (tzn. každý řádek a sloupec se dá jasně identifikovat a zjistit, co je v něm uloženo).

Třída „Unifier" unifikuje (sjednocuje) výstupy, jež mohou mít v různých situacích různou podobu. Pěkným příkladem je MAC adresa. Ta může mít různou podobu – např. operační systém Windows ji ukládá jako textový řetězec v podobě „`XX-XX-XX-XX-XX-XX`", zatímco Cisco zařízení v podobě „`XXXX.XXXX.XXXX`" (kde X je jedna číslice v šestnáctkové soustavě). Metoda *unifyMACAddress()* unifikuje tyto rozdílné přístupy do jednoho univerzálního.

Další velmi důležitý atribut komunikátoru je „deviceConnection". Ukládá objekt, vzniklý ze třídy „DeviceConnection", jehož úkolem je zajistit bezproblémové spojení se zařízením. Vzhledem k tomu, že se jedná o abstraktní třídu s abstraktními metodami, má potomka „TelnetConnection". Ten realizuje spojení s prvkem prostřednictvím protokolu *Telnet* (viz kapitola 4.1.1). Zde je velký potenciál rozšiřitelnosti do budoucna o další protokoly pro vzdálený přístup, zejména pro zabezpečený protokol *SSH*. Pomocí třídy „ConnectionResult" (a jejich potomků) lze identifikovat výsledek spojení.

Komunikátor si objekt třídy „DeviceConnection" opatřuje za pomoci třídy „DeviceConnectionFactory", postavené na návrhovém vzoru *Factory* (Továrna). Ta má pouze jednu statickou metodu, která vrací objekt třídy „DeviceConnectionFactoryResult", ve kterém se nacházejí atributy pro uložení chyby, výsledku a objektu třídy „DeviceConnection". Metoda pracuje tak, že se zkusí připojit pomocí protokolu *Telnet* (resp. pomocí třídy „TelnetConnection") k zařízení a výsledek vrátí jako zmíněný objekt. Pokud by ve vyvíjené aplikaci došlo k rozšíření protokolů pro vzdálený přístup, např. protokolu *SSH*, bude továrna fungovat tak, že se nejprve pokusí připojit pomocí protokolu *SSH* a pokud bude spojení neúspěšné[18], pokusí se spojit pomocí protokolu *Telnet*.

[18]Důvodů, proč by bylo spojení neúspěšné, může být víc. Může to být např. proto, že zařízení přes tento protokol neumí anebo nechce komunikovat, nebo zkrátka proto, že je zařízení nedostupné.



Obrázek 7: Objektový návrh části hledající cestu

5.4.2 Návrh grafové databáze

Návrh grafové databáze není potřeba demonstrovat. Grafová databáze je netypová, tudíž se návrh v praxi nedělá. Nicméně je vždy dobré dodržovat určitou strukturu, jinak mohou být data nekonzistentní a aplikace pracující nad nimi by mohla dělat problémy. Strukturu databáze v tomto případě určují databázové třídy, které byly již popsány a znázorněny v kapitole 5.1.3.

Je nutné zdůraznit, že aplikace počítá s tím, že databáze bude naplněna tak, jak je definováno v databázových třídách. Navíc pro správné nalezení cesty a následnou vizualizaci musí data v grafové databázi reflektovat reálný stav. Tím je myšleno mít správně uložená data o zařízeních, přihlašovacích údajích k zařízení, správné nastavení IP adres, VLAN atd. Pokud by byla databáze chybně naplněna, nelze počítat s tím, že by aplikace fungovala správně. Z toho, co teď bylo napsáno, je zřejmé, že pro **jednu konkrétní počítačovou síť** bude existovat **jedna konkrétní naplněná databáze**. Ukázky různě naplněných databází pro různé sítě (např. pro sítě demonstrované v kapitole určené pro testování) jsou součástí přílohy.

5.4.3 Návrh algoritmu pro nalezení cesty

Podstata backend aplikace spočívá v nalezení cesty mezi specifickým začátkem (bodem A) a koncem (bodem B). Jak je postupné nalezení cesty pomocí algoritmu v aplikaci realizováno, bude vysvětleno následovně.

Než algoritmus začne s hledáním, potřebuje znát zdrojový a cílový bod. To může být zjištěno různými způsoby (bylo to také probíráno v kapitole 5.4.1), např. pomocí názvů zařízení, doménových adres atd. V tomto případě byl zvolen způsob, spočívající ve vybrání zdrojového a cílového zařízení a následně jejich rozhraní (nabízeny jsou pouze ty, které jsou typu L3). Díky tomu, že se vyberou konkrétní rozhraní, je možno zjistit zdrojovou a cílovou IP adresu (rozhraní typu L3 má atribut pro uložení IP adresy). To zadá uživatel v části frontend a tím s ním interakce končí.

Dále je nutné zjistit první cílovou MAC adresu v cestě. Pokud se jedná o zdrojové zařízení typu PC, mohou nastat dvě situace – zdrojová a cílová IP adresa se nachází:

- ve stejné síti – pak první cílová MAC adresa bude zároveň konečnou cílovou adresou a algoritmus si ji zjistí od výchozí brány,
- v jiné síti – pak první cílová MAC adresa bude ta, jež je nastavená na rozhraní výchozí brány.

Výchozí brána se určí snadno, protože každé PC je zařazeno do jedné konkrétní VLAN a pro tu se, mimo jiné, uchovává rozhraní, reprezentující výchozí bránu (viz kapitola 5.1.3) a to jednoznačně určuje, ke kterému zařízení – výchozí bráně – patří[19].

[19]Přihlašovací údaje k zařízení a IP adresa, umožňující vzdálený přístup, jsou součástí každého zařízení ve formě atributů (samozřejmě pokud se jedná o síťový prvek, nikoliv PC).

U zdrojového prvku typu PC je nutné ještě zjistit první síťový prvek v cestě (většinou přístupový L2 přepínač). Ten je získán pomocí dotazu na grafovou databázi velmi jednoduše, protože každý prvek typu PC musí mít správně nastavené odchozí (tzv. „uplink“) rozhraní, které patří konkrétnímu (hledanému) zařízení.

Pokud by bylo zdrojové zařízení reprezentováno síťovým prvkem typu L3 (L3 přepínač či směrovač), první cílová MAC adresa se zjistí již na tomto zařízení tak, že se algoritmus pokusí najít ve směrovací tabulce záznam pro cílovou IP adresu a na základě výsledku se v ARP tabulce nalezne příslušná MAC adresa[20].

V posledním případě, kdy by bylo zdrojové zařízení reprezentováno síťovým prvkem typu L2 (klasický přepínač – zdrojové rozhraní by mohlo být realizováno jedině pomocí „SVI“), je opět nutné rozlišit dvě situace – jestliže se cílová adresa nachází:

- ve stejné síti – pak první cílovou MAC adresu algoritmus nalezne v ARP tabulce, které se dotáže na záznam pro cílovou IP adresu,
- v jiné síti – pak zařízení musí mít nastavenou IP adresu výchozí brány příkazem „`ip default-gateway <ip adresa>`“. Pokud příkaz chybí, hledání cesty končí s chybovým hlášením, v opačném případě se zjistí nastavená IP adresa výchozí brány a s ní je proveden dotaz na ARP tabulku.

Pro demonstraci principu algoritmu hledání cesty je určena následující síť (viz obrázek 8) s počátečním (PC-A) a koncovým (PC-B) prvkem, se síťovými zařízeními spojující tyto dva prvky[21] a celkem čtyřmi různými podsítěmi. IP adresa podsítě je vždy zobrazena pod obdélníkem, určujícím dosah dané podsítě. IP adresy zařízení jsou vždy napsány vedle nich, resp. vedle jejich portu[22].

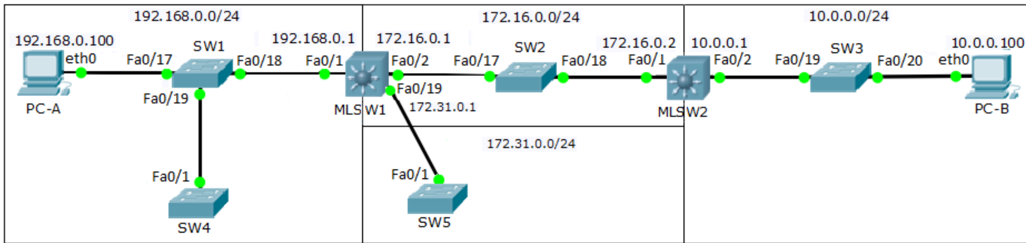
Algoritmus může začít, neboť má všechny potřebné údaje (první síťový prvek v cestě je „SW1“, IP adresa výchozí brány je „192.168.0.1“ a cílová IP adresa uzlu B je „10.0.0.100“) až na jeden – a tím je první cílová MAC adresa. Protože zdrojový prvek je typu PC, je nutné se připojit na zařízení, reprezentující výchozí bránu („MLSW1“) a cílovou MAC adresu zjistit tam. Zdrojová a cílová IP adresa nejsou ve stejné síti, tudíž se zjistí MAC adresa výchozí brány (rozhraní s IP adresou „192.168.0.1“). Vzhledem k tomu, že v průběhu hledání bude potřeba mít aktuální tabulky (zejména ARP a CAM), algoritmus provede na nejbližším L3 zařízení příkaz ping pro obnovení tabulek[23].

[20]Výsledek může být buďto „next-hop“ IP adresa nebo odchozí rozhraní – exit-interface. V případě next-hop IP adresy je dotaz na ARP tabulku daný touto IP adresou. V případě exit-interface je dotaz na ARP tabulku daný cílovou IP adresou, kterou algoritmus již zná.

[21]V síti jsou použity dva typy síťových zařízení – přepínače, pracující na vrstvě L2 s pojmenováním SWx (kde x reprezentuje číslo) a přepínače, pracující na vrstvě L3 s pojmenováním MLSWx (kde x reprezentuje číslo).

[22]IP adresu může mít přiřazenou pouze zařízení (resp. rozhraní), pracující na L3 vrstvě – to je v tomto případě PC nebo L3 přepínač.

[23]Pokud je zdrojový prvek typu PC, provede se příkaz ping na výchozí bráně, a to jak na zdrojovou, tak i na cílovou IP adresu. V jiném případě pouze na cílovou.



Obrázek 8: Demonstrativní síť algoritmu pro nalezení cesty

Každý síťový prvek, uložený v databázi, má atributy reprezentující IP adresu pro vzdálený přístup, přihlašovací údaje a mimo jiné i typ zařízení (L2/L3). Algoritmus tedy pokračuje tím, že si tyto informace pro první prvek v cestě extrahuje a spojí se s ním[24]. Po připojení k prvku mimo jiné provede dotazy pro získání informací (např. stavy portů), které budou později k vidění ve frontendové části. Tato data budou mít pro uživatele pouze informační charakter. Algoritmus ale hlavně potřebuje vědět další prvek v cestě, a protože ví, že je připojen na zařízení, pracující na L2 vrstvě, zobrazí si CAM tabulku, která řekne, přes jaký lokální port se dá dostat k MAC adrese výchozí brány (CAM tabulka vrátí konkrétně port „Fa0/18“). Jako další krok je potřeba zjistit, jaké zařízení je na druhé straně tohoto portu – na to slouží protokol CDP, jehož tabulka prozradí, které zařízení (resp. jeho název, tzv. hostname) se nachází na druhé straně (v tomto případě „MLSW1“). Samotný hostname postačí pro to, aby se v databázi nalezl příslušný prvek, neboť se předpokládá, že hostname je napříč sítí jednoznačný identifikátor síťového zařízení. Aplikace se tedy dotáže databáze na další prvek a opět se na něj spojí.

Nyní se jedná o L3 zařízení, algoritmus tedy pokračuje dotazem na směrovací tabulku, která určí, kudy se vydat k cílové IP adrese. Jsou dvě možnosti, které tabulka sdělí: buďto kterým odchozím rozhraním (tzv. exit-interface, v tomto případě tedy „Fa0/2“), nebo přes kterou vzdálenou IP adresu (tzv. next-hop IP adresa, zde konkrétně „172.16.0.2“), se k dané síti dá dostat. V případě odchozího rozhraní se sice ví „kudy ven“, neví se však, na jakou MAC adresu dalšího L3 zařízení směřovat. Prvek to řeší tak, že vyšle ARP dotaz s cílovou IP adresou. Zařízení, přes které se lze dostat na cílovou IP adresu, odešle ARP odpověď, kde sdělí MAC adresu svého portu, na kterou se má pokračovat. To znamená, že se pomocí ARP tabulky na základě cílové IP adresy zjistí cílová MAC adresa dalšího L3 zařízení v cestě. Protože se zná odchozí rozhraní, může se opět pomocí protokolu CDP zjistit přímo připojený prvek na druhé straně tohoto rozhraní. Ten se opět vyhledá v databázi a pokračuje se stejným způsobem dále (spojení se a vyhledání dalšího prvku v cestě).

[24]Pokud by u jakéhokoliv prvku v cestě nedošlo ke vzdálenému spojení, je uživateli na frontendu vrácena cesta až k takovému problémovému prvku a vypsána chybová hláška, popisující vzniklou situaci.

Tímto způsobem algoritmus iterativně pokračuje až do chvíle, kdy nalezne cílové zařízení, nebo nenajde v tabulkách záznam o dalším prvku v cestě (tím pádem je zřejmé, že se nachází chyba v konfiguraci prvku, která je posléze uživateli vypsána). Poté, co algoritmus dokončí hledání cesty, předá nalezené prvky (včetně sesbíraných dat) frontend části, která data zobrazí uživateli. Tím úloha algoritmu pro nalezení cesty končí.

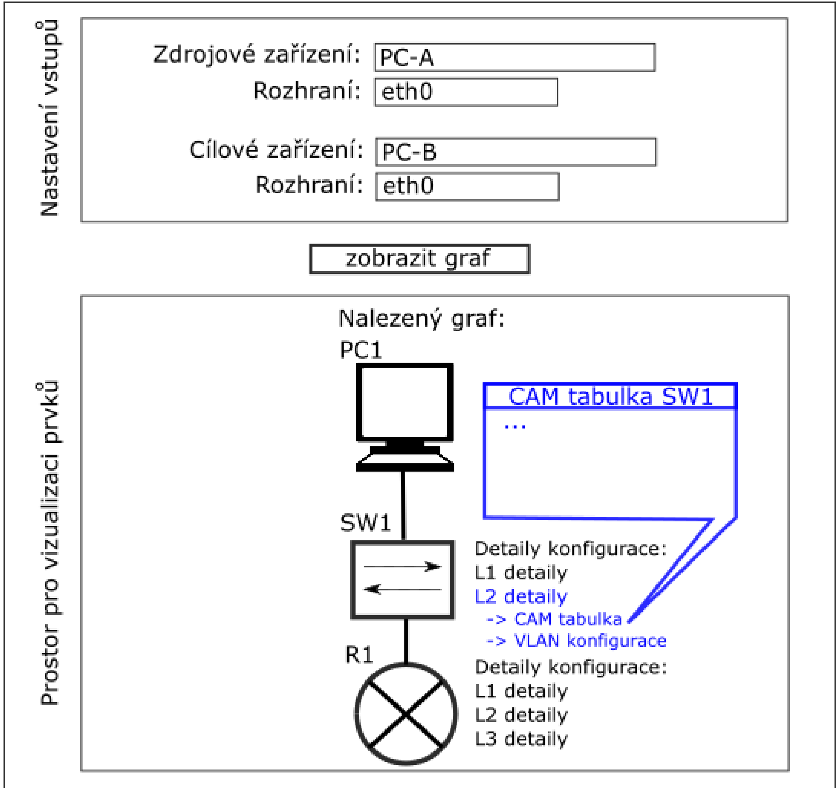
5.5 Frontend

Frontendová část vyvíjené aplikace má dva stěžejní úkoly:

- interagovat s uživatelem,
- zobrazovat výsledky hledání cesty.

Prvním bodem je myšleno zadání zdrojového a cílového bodu pro hledání cesty v počítačové síti a následné spuštění celého procesu, za který je odpovědná frontendová část. Jakmile je proces hledání ukončen, frontend přijme výsledek a zobrazí jej uživateli, o čemž pojednává druhý bod.

Samotný návrh spočívá zejména v grafickém rozložení webové stránky a je znázorněn na obrázku 9.

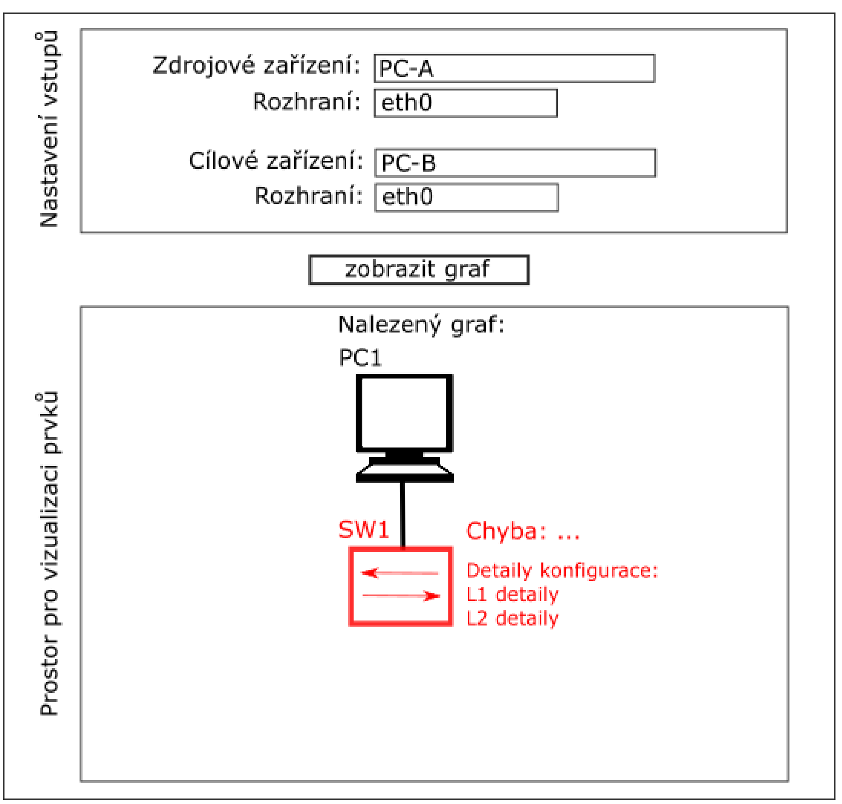


Obrázek 9: Grafický návrh frontendu s úspěšným nalezením cesty

Celé okno je rozdělené do dvou částí, které reflektují zmíněné úkoly frontendu. Vrchní část „Nastavení vstupů“ slouží pro interakci s uživatelem a umožňuje nastavit zdrojové a cílové zařízení. Jakmile je nějaké z nich vybráno, ihned se v položce „Rozhraní“ zobrazí dostupná rozhraní pro dané zařízení a je nutné udělat výběr. Na spuštění procesu hledání slouží tlačítko „zobrazit graf“.

Spodní část „Prostor pro vizualizaci prvků“ zobrazuje nalezenou cestu (resp. graf, kde uzly reprezentují zařízení a hrany spojení mezi nimi). U každého prvku je vypsaný jeho název, u síťových zařízení je navíc možnost zobrazit si detaily konfigurace na různých vrstvách (kliknutím na danou položku se zobrazí vyskakovací okno, jak je vidět na obrázku 9).

Pokud by se v nalézání cesty objevil problém, který znemožňuje pokračovat v průzkumu cesty, obrázek 10 značí, jak bude výsledek na webové stránce vypadat. Problémový prvek se zvýrazní odlišnou barvou a vypíše se také pravděpodobná chyba, bránicí pokračovat.



Obrázek 10: Grafický návrh frontendu s neúspěšným nalezením cesty

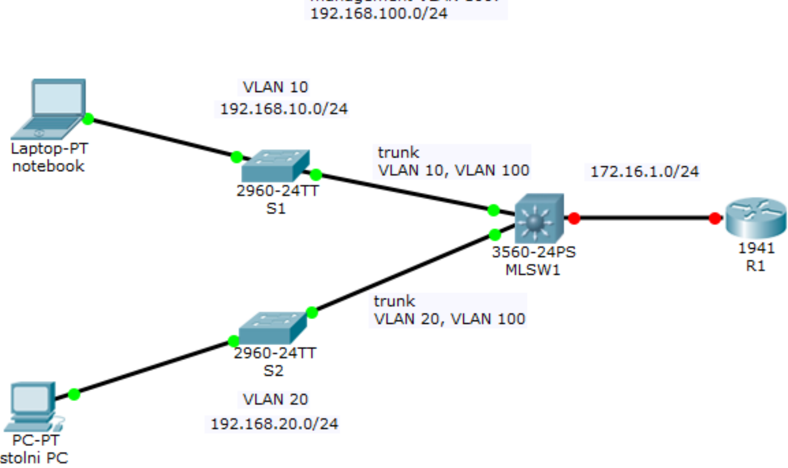
5.6 Testovací síť

Pro vývoj aplikace je nutné si připravit reálnou počítačovou síť tak, aby umožňovala efektivní testování v průběhu implementace. Na obrázku 11 je znázorněna použitá síťová topologie, složená z reálných zařízení[25], která byla k dispozici. Je navrhnuta tak, aby reflektovala problémy, které je potřeba v průběhu implementace řešit. V topologii jsou obsažené důležité vlastnosti:

1. ke každému zařízení je možno se vzdáleně připojit,
2. existuje počáteční bod A a koncový bod B,
3. existují všechny důležité typy zařízení (L2 i L3 přepínač, směrovač),
4. je prováděno přepínání a směrování paketů,
5. existuje cesta (z „PC“ na směrovač „R1“), která umožňuje testovat nalezení cesty přes více L3 zařízení,
6. je zde prvek, který může současně pracovat jak na úrovni L2, tak i na L3 (vícevrstvý přepínač „MLSW1“).

Z výčtu je zřejmé, že je pokryta většina scénářů, které mohou nastat při hledání cesty. Ovšem najdou se i situace, které v takové síti nelze bez zásahů do konfigurace simulovat. Může se jednat o nalezení cesty mezi prvky typu PC ve stejné síti (ať už přepínáním paketů pouze na úrovni L2 přepínačů, tak i prostřednictvím L3 přepínačů, které by v danou chvíli pracovaly na úrovni L2).

Kompletní konfigurace každého zařízení jsou součástí přiloženého CD.



Obrázek 11: Návrh testovací sítě pro účely implementace

[25]Konkrétně byly použity tři typy zařízení společnosti Cisco – přepínač, pracující pouze na vrstvě L2 (model 2960, 8TC-L), přepínač s možností práce na L3 (model 3560, 8PC-S) a směrovač (model 1921/SEC-K9).

6 Implementace aplikace

Implementaci lze vnímat jako zhmotnění návrhu do programového kódu. Bez ní by návrh ztrácel smysl, a protože se z něj vychází, budou využívány třídy a metody, navržené v kapitole 5.

Vzhledem k objemu implementovaných věcí nelze popsat kompletní implementaci. Proto zde budou představeny stěžejní a nejzajímavější bloky, včetně ukázky programového kódu (tři tečky v kódu značí neúplné vypsání kódu).

Kompletní zdrojové kódy lze najít v elektronické příloze.

6.1 Implementace úprav části CORE v existujícím projektu

O navržených úpravách pojednává kapitola 5.1.3. V ní byl UML objektový návrh rozdělen do dvou částí. Nejvíce změn se odehrálo v druhé části (viz obrázek 3), tudíž všechny ukázky implementace budou vybírány z ní.

Některé navrhované změny jsou si velmi podobné, například třída „DeviceConfiguration“ a „Credentials“ (včetně jejich repositářových tříd a služeb) se liší pouze svými atributy. Proto i jejich implementace jsou si velmi podobné. V takových případech bude předložena implementace pouze jedné ze změn.

Kompletní implementaci jak existujících tříd, tak nově přidaných či upravených, lze najít v přílohách, konkrétně v balíčku „domain“ složky „app-backend/core“. Jsou v něm implementované třídy databázové, repositářové a třídy služeb.

6.1.1 Databázové třídy

Ukázka zdrojového kódu se týká databázové třídy „Credentials“, která byla nově přidaná.

```
@Entity
public class Credentials extends EnumType<Credentials> {
  private String userLogin;
  private String userPassword;
  private int privLevel;
  private String enablePassword;
  ...
```

Anotace `@Entity` se používá u definování databázových tříd pro *in-memory* databázi. Třída dědí abstraktní třídu `EnumType`, která zabezpečuje například správné přidělení ID v databázi.

Ve třídě existuje několik konstruktorů. Následná ukázka zahrnuje konstruktor jak bez parametrů, tak s parametry, reprezentující všechny dříve definované atributy třídy. Ve většině případů se využívá konstruktor se vstupními parametry.

Metoda `super(title)` zavolá konstruktor předka, čímž se zajistí správné přidělení ID[26] v databázi.

```
...
protected Credentials(){ super(); }

public Credentials(String title, String userLogin, String userPassword,
    int privLevel, String enablePassword){
   super(title);
   this.userLogin = userLogin;
   this. userPassword = userPassword;
   this.privLevel = privLevel;
   this.enablePassword = enablePassword
}
...
```

6.1.2 Třídy implementující služby

Pro potřeby vizualizace je ze všech tříd tohoto typu využívána pouze třída „DeviceSimpleService", která obsahuje stěžejní metodu `findPathFromAtoB(...)`, která spouští proces hledání cesty a vrací všechna nalezená zařízení v cestě.

```
@Service("DeviceService")
@Transactional
class DeviceSimpleService extends ... implements DeviceService {
   ...
   public Iterable<Device> findPathFromAToB(Long srcDevIntId, Long
       dstDevIntId){
      PathFinder pathFinder = new PathFinder(...);
      return pathFinder.findPathByDeviceInterfaces(srcDevIntId,
         dstDevIntId);
   }
   ...
```

Třída implementuje rozhraní `DeviceService`, tudíž i všechny jeho definované metody, včetně zmíněné. Metoda `findPathFromAtoB(...)` vytváří objekt třídy `PathFinder`, starající se o celý proces hledání (třída byla popsána v kapitole 5.4.1). Její parametry reprezentují zdrojový a cílový uzel hledané cesty. Návratová hodnota metody se zajistí kódem, začínajícím klíčovým slovem `return`, za pomoci metody `findPathByDeviceInterfaces(...)` zmíněného objektu.

[26]ID se v tomto případě generuje z atributu `title`. Nicméně způsob, jakým je prováděno vygenerování, je definován ve třídě „EnumType" a dá se kdykoliv změnit.

6.1.3 Repositářové třídy

Repositářové třídy jsou v rámci hledání cesty využívány všechny. Bez nich by se nedalo přistupovat k datům v databázích. Jejich metody[27] vždy vrací objekt, definovaný příslušnou databázovou třídou, který se nalezne prostřednictvím specifického databázového dotazu (viz následná ukázka kódu).

Některé databázové třídy ovšem nemají uvedeny žádné metody (např. třída „DeviceConfigurationRepository“, viz obrázek 3). Avšak tím, že vždy dědí „základní“ databázovou třídu, přebírají automaticky i všechny její metody (přinášející *CRUD* operace), tudíž i tak jsou zajištěné základní operace s daty.

```
@Neo4jRepository
public interface DeviceRepository extends GraphRepository<Device> {
   ...
    @Query("MATCH (d:Device) WHERE d.title = {hostname} MATCH
        p=(d)-[*0..1]-(m) RETURN p")
   Device findByHostname(@Param("hostname") String hostname);
   ...
```

Anotace `@Neo4jRepository` určuje, že se jedná o repositářovou třídu grafové databáze *Neo4J*. Zmíněnou základní databázovou třídou je `GraphRepository`. Ukázka znázorňuje metodu `findByHostname(...)`, která má najít zařízení na základě atributu `hostname`, jenž je nastavený ve vstupních parametrech. Anotace `@Param` pouze propojuje vstupní parametry s dotazem, definovaným v anotaci `@Query`.

Ve skutečnosti databázové třídy nejsou JAVA třídami, ale rozhraními (klíčové slovo *interface*). To znamená, že by jejich metody měly být implementovány jinou třídou. Nicméně anotace `@Query` zajistí potřebnou implementaci a sdělí, co se má v případě zavolání metody vykonat. V závorkách této anotace se vždy nachází databázový dotaz, pro grafové databáze to vždy bude *Cypher* dotaz (viz ukázka kódu).

6.2 Implementace úprav části API v existujícím projektu

O navržených změnách v části *API* pojednává kapitola 5.1.4. Objektový návrh, který značí přidané či změněné věci, je vyobrazen na dvou dílčích obrázcích (obr. 4 a 5), tvořících jeden celek.

Změny se týkají tříd typu Kontrolér, Zdroj a Překladač zdrojů. Opět se v návrhu objevuje více tříd stejného typu, které mají podobnou strukturu, a tedy i implementaci. Bude proto vybrán od každého typu jeden vhodný reprezentant, na kterém se provede ukázka implementace.

Kompletní implementaci všech tří typů lze opět najít v přílohách, konkrétně v balíčku „url“ složky „app-backend/api“. Balíček je dále členěn na další balíčky,

[27] Všechny explicitně definované metody v repositářových třídách mají za úkol zpracovat data neobvyklým způsobem. Může se jednat například o čtení dat prostřednictvím specifického dotazu.

jejichž název reflektuje název databázových tříd. Jsou v něm implementované třídy databázové, repositářové a třídy služeb.

6.2.1 Třídy typu Kontrolér

Kontroléry zajišťují komunikaci s frontendem. Obsahují metody, jež jsou volány na základě *HTTP* požadavku, který je odeslán na příslušnou *URL*, např. prostřednictvím technologie *AJAX* (nebo také za pomoci internetového prohlížeče, pokud k *API* přistupuje člověk). Namapování *URL* na kontrolér realizuje sama třída. Podoba může být například taková:

```
@RequestMapping(path = Rest.REST + "/devices") // REST = "/api"
...
public class DeviceController {
...
```

Pro účely vizualizace sítě si frontend vyžádá nalezení cesty prostřednictvím kontroléru „DeviceController“. O data žádá na *URL*:
`http://localhost:8080/api/devices/findPath/{ID zdroje}/{ID cíle}`.
Cíl `localhost:8080` je nastaven v případě, že backend běží na lokálním serveru se síťovým portem 8080. ID zdroje a cíle se nahrazují za čísla, reprezentující databázová ID zdrojového a cílového rozhraní. Namapování *URL* s příslušnou metodou je znázorněno v ukázce:

```
...
@RequestMapping(path = "/findPath/{srcDevIntId}/{dstDevIntId}",
          method = RequestMethod.GET)
  public ResourcesPage<DeviceResource> findPath(@PathVariable Long
     srcDevIntId, @PathVariable Long dstDevIntId) throws
     ResponseException {
  Iterable<Device> deviceIterable =
     deviceService.findPathFromAToB(srcDevIntId, dstDevIntId);
  if(deviceIterable == null) throw new NotFoundException("Devices not
     found by these interfaces.");
  return deviceResourceAssembler.toResourcesPage(deviceIterable);
  }
...
```

Namapovaná metoda se nazývá `findPath()`, s parametry zdrojového a cílového ID rozhraní. Anotace `@PathVariable` značí, že se parametry nacházejí v *URL*. Text za ní (`Long`) určuje, o jaký datový typ se jedná a kde přesně se parametry v *URL* vyskytují.

Hlavní krok spočívá v zavolání metody `findPathfromAToB(...)` objektu `deviceService` (jedná se o objekt třídy typu Služba), jež byla již probrána v této kapitole, části 6.1.2. Metoda vrací objekt `Iterable<Device>`, což je posloup-

nost zařízení v cestě. Pokud by nic nevrátila (resp. by vrátila `null`), vyvolala by se výjimka `NotFoundException`, popisující tento stav ve svém parametru. Návratová hodnota se zajistí zavoláním metody `toResourcesPage(...)` objektu `deviceResourceAssembler` (jedná se o objekt třídy typu Překladač zdrojů). Výslednou návratovou hodnotou je stránka zařízení ve formátu *JSON*. Výsledek může mít takovou podobu:


```
{
  "content": [
    {
      "id": 49,
      "deviceTypeId": "Switch",
      "title": "sw-a-3",
      "manageIPId": 42,
      ...
      "links": [
        {
          "rel": "interfaces",
          "href": "http://localhost:8080/api/devices/48/interfaces"
        },
        ...
      ]
    },
  ...
```


6.2.2 Třídy typu Zdroj

Třídy tohoto typu definují, **co** je možno zařadit do výsledného výstupu ve formátu *JSON*. V zásadě pouze reflektují atributy databázové třídy.

Vhodným reprezentantem pro zobrazení implementace může být třída „IPv4AddressResoure“. Jedná se o třídu typu Zdroj pro ukládané IP adresy verze 4 v grafové databázi.

```
@ApiModel(description = "Informace o IPv4 adrese")
@JsonPropertyOrder({"id", "address", "netmask"})
public class IPv4AddressResource extends ResourceSupport {
  @ApiModelProperty(readOnly = true, value = "jedinecny identifikator")
  @JsonProperty("id")
  public Long id;
  @ApiModelProperty("adresa")
  public String address;
  @ApiModelProperty("maska")
  public String netMask;

}
```

Třída je potomkem třídy `ResourceSupport`, která přináší užitečné funkcionality, využívané ve třídách typu Překladač zdrojů. Umožňuje například využít metodu, která do výsledného *JSON* zakomponuje odkazy (tzv. linky) na jiné objekty.

Anotace `@JsonPropertyOrder(...)` specifikuje, v jakém pořadí mají být jednotlivé atributy seřazeny. Nejdůležitější částí je specifikace atributů, které bude výsledný *JSON* obsahovat. Anotace `@ApiModelProperty(...)` značí, že se jedná o atribut. Její parametr má pouze informativní charakter.

6.2.3 Třídy typu Překladač zdrojů

Zatímco třídy typu Zdroj určují, **co** může být zařazeno do formátu JSON, třídy typu Překladač zdrojů popisují, **jak** bude určitý databázový objekt složen do formátu *JSON*. Překladače zdrojů tedy využívají zdroje a vytváří výsledný objekt ve zmíněném formátu.

V části 6.2.2 popisovala implementaci třída pro IP adresy verze 4. Aby bylo zřejmé, jak překladač zdrojů dělá svou práci, reprezentuje třídy tohoto typu opět třída pro IP adresy verze 4.

```
public class IPv4AddressResourceAssembler extends
    AbstractResourceAssembler ... {
   ...
   @Override
   public IPv4AddressResource toResource(IPv4Address entity) {
      try {
         IPv4AddressResource resource = new IPv4AddressResource();
         resource.id = entity.getId().value;
         resource.address = entity.getAddress();
         resource.netMask = entity.getNetMask();
         resource.add(linkTo(methodOn(ipv4AddressController)
                     .getOneById(entity.getId()))
                     .withSelfRel());
         return resource;
      } catch (Exception e){
         throw new IllegalStateException("Transform entity to resource
             failed.", e);
      }
   }
   @Override
   public IPv4Address toEntity(IPv4AddressResource resource){
      return new IPv4Address(
            resource.address,
            resource.netMask);
      }
   }
}
```

Třída dědí třídu `AbstractResourceAssembler`, která přináší užitečné metody. Jedna z nich zde již dříve byla použita v kontroléru (kap. 6.2.1), konkrétně metoda `toResourcePage(...)`, která vytvořila „stránku“ zařízení (tedy výstup ve formátu *JSON* s více objekty)

Ukázka značí, že se překrývají[28] metody `toResource(...)` a `toEntity(...)`. První vytváří z objektu databázové třídy objekt třídy typu Zdroj, druhá dělá přesný opak. Využívá se zde zmíněná metoda, vytvářející odkazy (linky) ve výsledném objektu (viz kapitola 6.2.2). Konkrétně se vytváří odkaz směřující „sám na sebe“ (`.withSelfRel()`) pomocí metody `add(...)`.

6.3 Implementace hledání cesty

Předchozí sekce popisovaly implementace věcí, které rozšiřovaly existující projekt proto, aby šlo implementovat aplikační část, starající se o nalezení cesty.

Tato kapitola pojednává o implementaci části aplikace, jež je zodpovědná za nalezení cesty. Objektový návrh pro tuto část byl představen v kapitole 5.4 na obrázku 5.4.1. Každá následující sekce se zabývá specifickou částí objektového návrhu, zodpovědnou za určitou funkcionalitu.

Veškeré zdrojové kódy lze najít v přílohách, konkrétně v balíčku „deviceconnection“ složky „app-backend/core“. Balíček obsahuje kompletně všechny třídy, které se starají za nalezení cesty.

6.3.1 Vzdálený přístup k zařízení

V kapitole 4.2.2 bylo zmíněno, že pro komunikaci s prvkem se využívá knihovna *Apache Commons Net*, konkrétně její třída „TelnetClient“, jenž přináší fungující protokol *Telnet*. Využití je jednoduché, jak značí ukázka kódu:

```
...
import org.apache.commons.net.telnet.TelnetClient;
...
public class TelnetConnection extends DeviceConnection {

  private final TelnetClient telnetClient;

  private PrintWriter out;
  private InputStreamReader in;
  ...
```

Knihovna se připojí pomocí klíčového slova `import`. Daly by se připojit všechny třídy, které knihovna nabízí, záměnou kódu `.TelnetClient` za `.*`. Vzhledem k tomu, že se opravdu využívá pouze třída `TelnetClient`, není to však nutné (navíc

[28]Překrytím metody předka se vytváří její nová, odlišná implementace.

se nepatrně zrychlí celkové sestavení aplikace, tzv. *build*). Po tomto kroku lze již vytvářet objekty z této třídy a používat jejich metody.

Komunikace se zařízením probíhá prostřednictvím příkazů. Ty se posílají výstupním kanálem (proměnná `out`). Odpovědi na příkazy posílá zařízení naopak vstupním kanálem (proměnná `in`).

Celý proces navázání komunikace se zařízením není nijak složitý díky využití knihovny. Objekt `telnetClient` nabízí metodu `connect(ip)`, která vyžaduje v parametrech identifikaci objektu (např. doménové jméno či IP adresa). Výsledek spojení lze zjistit pomocí metody (tzv. *getteru*) `isConnected()`.

```
...
@Override
public ConnectionResult connect(String ip, String login, String pass) {
   try {
      telnetClient.connect(ip);
      if (telnetClient.isConnected()) {
         initCommunication();
      } else {
         return new NotConnectedResult();
      }
      return login(login, pass);
   } catch (IOException ex) {
      return new ConnectionResult(ex.getMessage());
   }
}
...
```

Pokud se aplikace úspěšně spojí, zavolá se metoda `initCommunication()`, která pouze vytvoří objekty pro proměnné `in` a `out`. V opačném případě metoda `connect(...)` vrátí objekt, reprezentující neúspěšné spojení včetně chyby.

Pro funkční komunikaci nestačí samotné navázání spojení. Je nutné se ještě přihlásit do příkazové řádky. O to se pokusí metoda `login(...)`, jejíž parametry reprezentují přihlašovací údaje.

```
...
private ConnectionResult login(String login, String pass) {
   out.println(login);
   String result = sendCommand(pass);
   if(result.contains("invalid")) return new InvalidCredentialsResult();
return new ConnectionResult();
}
...
```

Zařízení po spojení požaduje uživatelské jméno („username“) a čeká na odpověď, kterou metoda pošle příkazem `out.println(login)`. Dále se zadá hes-

lo metodou `sendCommand(pass)`, která vrací text, reprezentující odpověď prvku na pokus o přihlášení. Díky tomu lze analyzovat, na základě obsahu vráceného textu, zdali to bylo úspěšné či nikoliv (což určují vrácené objekty `return new ConnectionResult()` nebo `return new InvalidCredentialsResult()`).

Následné zasílání příkazů a přijímání odpovědí zajišťuje zmíněná metoda `sendCommand(příkaz)`. Její implementace se stará o správné čtení ze vstupu (`in`). Zařízení posílá odpověď na příkazy po znacích, tudíž je nezbytné každý znak zpracovat a analyzovat (zdali neindikuje konec odpovědi atd.). Vzhledem k tomu, že implementace této metody je dlouhá a relativně náročná na vysvětlení, nebude zde uváděna. Pro případné zájemce lze implementovanou metodu najít ve zdrojových kódech v příloze (balíček „connections“).

6.3.2 Komunikátor a jeho potomci

Komunikátor zajišťuje dolování dat ze zařízení, nezbytných k nalezení cesty. Může se jednat například o zjištění sousedního prvku na základě protokolu *CDP* či o nalezení MAC adresy k určité IP adrese pomocí *ARP* tabulky. Kromě toho se komunikátor stará o vytvoření objektu pro komunikaci s prvkem (viz předešlá kapitola).

```
...
public abstract class DeviceCommunicator {
  //atributy
  protected DeviceConnection deviceConnection;
     ...
  //metody
  public DeviceCommunicatorResult connectToDevice(){
     ...
  }
  ...
```

Ukázka ilustruje definici třídy `DeviceCommunicator`, jeden příklad atributu (do nějž se ukládá zmíněný objekt, poskytující vzdálený přístup k prvku) a metody (která se pokusí získat objekt pro komunikaci).

Jedná se o abstraktní třídu, tudíž se předpokládá výskyt potomků. Obrázek 7 znázorňuje, že potomci opravdu existují. Každý z nich mění implementaci určitých metod tak, aby reflektovaly funkcionalitu L2 (`L2Communicator`) resp. L3 (`L3Communicator`) zařízení, nebo i přidávají metody nové.

Pro ilustraci dolování dat prostřednictvím komunikátoru slouží následující ukázka metody, která zjišťuje odchozí rozhraní (exit-interface) na základě cílové MAC adresy, za pomoci *CAM* tabulky. Metoda je implementována ve třídě `L2Communicator`.

```
...
public String getExitInterface(String MACAddress){
  String result = sendCommand("show mac address-table | include " +
     MACAddress);
  if (result.trim().equals("") || result == null) return null;
  GenericTable table = this.dataParser.parseCAMTable(result);
  for(ArrayList<String> row : table) {
    if(row.get(0).equals(MACAddress)) return row.get(1);
  }
return null;
}
...
```

Na začátku metody se zasílá na zařízení příkaz, který se dotazuje na *CAM* tabulku pro konkrétní MAC adresu, určenou ve vstupním parametru metody. Poté se ověří, bylo-li tímto příkazem něco nalezeno – kdyby ne, metoda vrátí `null`. Pokud se nějaká odpověď (v proměnné `result`) získala, využije se metoda objektu `dataParse`, která zpracuje získanou *CAM* tabulku. Vrací objekt třídy `GenericTable`, která byla představena a vysvětlena v kapitole 5.4.

Uloženou *CAM* tabulku prostřednictvím třídy `GenericTable` lze již velmi jednoduše využívat. Vzhledem k tomu, že tabulka má přesně danou strukturu, může se hledat shoda s MAC adresou v konkrétním sloupci (`if(row.get(0).equals(MACAddress))`). Pokud se nalezne shoda, metoda skončí a vrátí odchozí rozhraní, umístěné ve druhém sloupci.

6.3.3 Parsování dat

O parsování dat se stará třída „DataParser“. Všechny její metody slouží pro zpracování konkrétních tabulek, které se získaly zasláním příkazu na zařízení. Vstupní hodnotou je vždy textový řetězec. Vzhledem k tomu, že každá metoda je vyhrazena na konkrétní tabulku, ví, jakou má strukturu a jak ji zpracovat. Výstupní hodnotou je objekt „GenericTable“, která řádky a sloupce tabulky, dříve jako jeden prostý textový řetězec, ukládá jako matici.

Atributy třídy jsou statické proměnné (resp. konstanty), které určují pro každou tabulku její hlavičku. To proto, aby určitá metoda věděla, kdy v textu začíná samotná tabulka.

Kompletní zdrojové kódy tříd, které slouží pro parsování dat, lze najít v přílohách (balíček „dataparse“). V ukázce kódu předešlé kapitoly byla využita metoda `parseCAMTable(...)`, ale bez ukázky implementace. Na to je prostor nyní.

```
...
public GenericTable parseCAMTable(String rawData){

ArrayList<ArrayList<String>> tempReturn = new ArrayList<>();
String[] trimParseTable = rawData.trim().split("\r\n");

for(String row : trimParseTable) {

   ArrayList<String> tempColumn = new ArrayList<>();
   tempColumn.addAll(Arrays.asList(row.split(WHITESPACE + WHITESPACE)));
   tempColumn.removeIf(new EmptySpacePredicate());

   String firstCol = tempColumn.get(1).trim();
   String secondCol = Unifier.unifyNetInterface(tempColumn.get(3).trim());

   ArrayList<String> finalColumns = new ArrayList<>();
   finalColumns.add(firstCol);
   finalColumns.add(secondCol);

   tempReturn.add(finalColumns);
}
return new GenericTable(tempReturn);
}
...
```

Proměnná `trimParseTable` (datového typu textového pole) vznikne tak, že se vstupní (tzv. surová) data rozdělí na jednotlivé řádky pomocí metody `split()`. Hodnoty (řádky) proměnné pak lze jednoduše procházet pomocí cyklu `for(...)`. V něm se při každém průchodu vytvoří nová proměnná (`tempColumn`), umožňující ukládat jednotlivé sloupce procházeného řádku.

Proměnná `tempColumn` se naplní opět metodou `split()`, která jednotlivé sloupce řádku rozpozná tak, že jsou mezi nimi alespoň dvě mezery. Poté se uloží pouze potřebné sloupce, a to do dvou dočasných proměnných (`firstCol` a `secondCol`). U druhé proměnné se využívá statická metoda třídy `Unifier`, která unifikuje odchozí rozhraní, získané z *CAM* tabulky. Oba sloupce daného řádku jsou na konci cyklu uloženy do výstupní proměnné.

6.3.4 Nalezení cesty

Celý proces, starající se o nalezení cesty, řídí třída „PathFinder“. K tomu využívá jak třídy z existujícího projektu (včetně rozšiřujících změn a přidaných tříd), tak probraný komunikátor (třída „DeviceCommunicator“).

Třída obsahuje spoustu privátních metod, které přinášejí určitou funkcionalitu, potřebnou při nalézání cesty. Jednu z nich zobrazuje následující ukázka:

```
...
private boolean areAddressesInSameSubnet(IPv4Address srcAddress,
    IPv4Address dstAddress){
   SubnetUtils subnetUtils = new
      SubnetUtils(srcAddress.getAddress(),srcAddress.getNetMask());
   return subnetUtils.getInfo().isInRange(dstAddress.getAddress());
}
...
```

Metoda `areAddressesInSameSubnet(...)` umožňuje zjistit, zdali se zdrojová IP adresa (včetně masky sítě) nachází ve stejné síti jako cílová. K tomu je využívána třída `SubnetUtils`, která je součástí knihovny *Apache Commons Net*. Ta zpřístupňuje metodu `isInRange(...)`, která provede porovnání IP adres a vrátí výsledek datového typu `boolean` (pravda/nepravda).

Celý proces hledání spouští metoda `findPathByDeviceInterfaces(...)`. V ní jsou nastaveny všechny potřebné počáteční údaje pro algoritmus hledání cesty (viz kapitola 5.4.3).

```
...
public Iterable<Device> findPathByDeviceInterfaces(Long srcDevIntId, Long
    dstDevIntId){
   this.device = deviceRepository.getDeviceByInterfaceId(srcDevIntId);
   this.dstDevice = deviceRepository.getDeviceByInterfaceId(dstDevIntId);
   ...
   IPv4Address srcAddress = findDevIntIPv4Address(srcDevIntId);
   ...
   this.srcIP = srcAddress.getAddress();

   IPv4Address dstAddress = findDevIntIPv4Address(dstDevIntId);
   ...
   this.dstIP = dstAddress.getAddress();
   ...
   if(areAddressesInSameSubnet(srcAddress, dstAddress)){
      this.dstMAC = obtainFirstMAC(srcDeviceInterface, true);
   } else{
      this.dstMAC = obtainFirstMAC(srcDeviceInterface, false);
   }
   ...
   findPath();
   return deviceIterator;
}
...
```

Ukázka obsahuje části kódu metody, která spouští proces hledání na základě zdrojového a cílového rozhraní. Jsou zde znázorněny ty části, které jsou zodpovědné

za nastavení potřebných údajů (zdrojový a cílový prvek, IP adresa a první cílová MAC adresa atd., viz kapitola 5.4.3).

Předposlední krok zavolá metodu `findPath()`, která započne hledání na síťových prvcích. Je implementována tak, jak popisuje algoritmus pro hledání cesty. Implementace je velmi dlouhá a na kompletní předložení a vysvětlení kódu zde není prostor, takže bude vyňata pouze jedna část z celé metody.

```
if(this.device.getDeviceType().value.toLowerCase()
                      .equals(DEVICE_TYPE_L2)){
   ...
   String exitInterface =
       deviceCommunicator.getExitInterface(getDstMAC());
   ...
   String neighborDevice =
       deviceCommunicator.getNeighborDevice(exitInterface);
   ...
   Device nextDevice = deviceRepository.findByHostname(neighborDevice);
   ...
}
...
```

Část ilustruje hledání dalšího zařízení v cestě na prvku, pracujícím na úrovni L2 (část `equals(DEVICE_TYPE_L2)`). Nejdříve se zjistí v *CAM* tabulce odchozí rozhraní na základě cílové MAC adresy, pomocí metody `getExitInterface(...)` komunikátoru. Poté se metoda `getNeighborDevice(...)` pokusí nalézt „hostname“ sousedního prvku pomocí *CDP* tabulky, který se nachází na druhé straně odchozího rozhraní. Díky tomu už se za pomoci repositáře pro zařízení lehce nalezne další prvek v cestě.

6.4 Frontend

Frontend část aplikace se stará o interakci s uživatelem. Podrobněji o této části pojednává kapitola 5.5. Existuje několik souborů, které se starají o chod celé části:

- *HTML* soubor, který se stará o rozložení a zobrazení webové stránky,
- *CSS* soubor, který se stará o vizuální styl stránky,
- *JavaScript* soubor, který se stará o funkční část frontendu.

Celou frontend část, včetně všechy zmíněných souborů, lze najít v příloze ve složce „app-frontend“.

6.4.1 HTML soubor

Soubor je psán ve značkovacím jazyku *HTML*. Jeho hlavní úlohou je pospojovat k sobě další nezbytné soubory pro správné fungování frontendu.

```
<html>
   <head>
      ...
         <link rel="stylesheet" href="style.css" />
         <script src="app.js"></script>
         <script
            src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
         ...
    </head>
    ...
</html>
```

Ukázka zahrnuje nejdůležitější pasáže části `body`. Ty připojují soubory *CSS*, *JavaScript* a v neposlední řadě také knihovnu *jQuery*.

Zbytek souboru definuje rozložení stránky přesně tak, jak znázorňuje obrázek 9. Nejdříve vytvoří formulář pro zadávání vstupních informací:

```
...
<form id="device-form">
   ...
   <button type="submit">Zobrazit graf</button>
   ...
</form>
...
```

V něm je vytvořeno mimo jiné také tlačítko, jenž slouží jako spoušť pro hledání cesty. Jakmile se stiskne, zavolá se příslušná operace v *JavaScript* souboru (viz kapitola 6.4.3).

6.4.2 CSS soubor

Podle Martina Malého (2010) jsou kaskádové styly „dnes až na výjimky přijímány jako standard pro popis vizualizace formátu hypertextových dokumentů.“ Existuje však knihovna, nesoucí název *LESS*, která do *CSS* zavádí užitečné vlastnosti. Ta byla využita i pro potřeby frontendu. Kompletní *LESS* soubor („style.less“) lze najít v přílohách.

Jsou zde definovány styly pro různé části stránky. Pro účely demonstrace bude vybraná pouze část implementace, která formátuje graf, reprezentující nalezenou cestu.

```
...
#graph {
  ...
  .node.error .img {background-color: rgba(255,0,0,0.50);}
  .node {
    min-height: 200px;
    ...
    .img {
      width: 200px;
      height: 200px;
      ...
    }
  }
}
...
```

V případě chyby se obarví pozadí obrázku, reprezentující nalezený prvek, (`.node.error .img`) dočervena, jinak se formátování řídí tím, co je uvedeno v těle `.node`, resp. `.img`.

6.4.3 JavaScript soubor

Soubor zajišťuje komunikaci s backendem. Stará se například o načtení dostupných zařízení, včetně jejich rozhraní, do polí pro zadávání vstupních údajů.

```
...
$(document).ready(function () {
  downloadDevices();
  $("#source-device-id, #destination-device-id").change(function () {
    updateInterfacesSelects($(this).val());
  });
  ...
}
...
```

S backendem si vyměňují informace prostřednictvím technologie *AJAX*. Jeden příklad ilustruje následující ukázka, na níž je znázorněn kus vytvořené funkce `downloadDevices()`:

```
$.ajax({
  url: "http://localhost:8080/api/devices",
  method: "GET",
  success: function (data) {
        ...
  },
  error: function (data) {
        ...
  }
});
```

AJAX se pokouší připojit na *URL* `http://localhost:8080/api/devices` backend části *API* (viz sekce 6.2.1). Požadavek *HTTP* využívá metodu `GET`, která podle Dostálka (2005, str. 463) slouží k "dotazování klienta na konkrétní informace uložené na serveru." Jakmile dojde odpověď od backendu, vyvolá se tzv. „call-back“ funkce. Jestli se bude jednat o funkci, definovanou v části `success` nebo `error`, už rozhoduje úspěšnost *AJAX* požadavku.

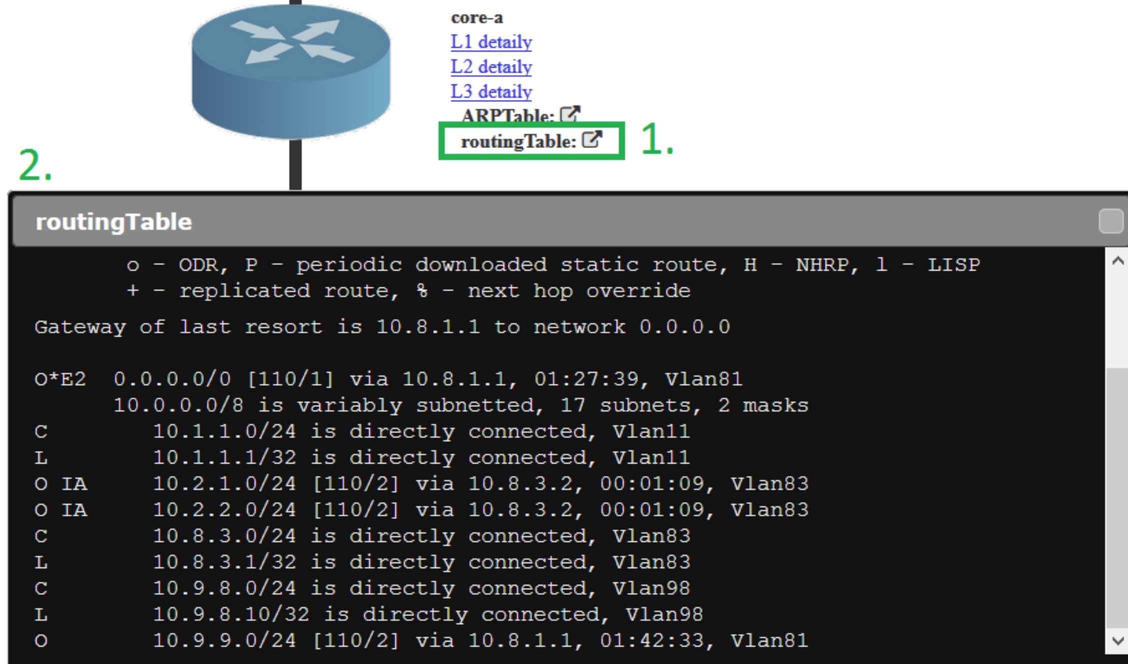
Knihovna *jQuery* nabízí mnohem víc než jen *AJAX*. Její odnož, jež se nazývá *jQuery UI*, obsahuje množinu funkcí, které slouží pro interakci s uživatelem. Jejich připojení k aplikaci je velmi jednoduché, stačí pouze přidat jeden řádek kódu:

```
<script src="https://code.jquery.com/ui/1.12.1/jquery-ui.min.js"></script>
```

Tím se zpřístupní řada objektu, jedním z nich je např. „Dialog“ (Dialog, 2017), který umožňuje vytvářet dialogová okna s informacemi. Frontend jej využívá pro zobrazování konfiguračních informací o zařízení pro uživatele. Implementace dialogového okna je relativně jednoduchá. Stačí použít nově nabízené funkce. Tou je např. `dialog()`, která umožňuje vytvořit objekt, reprezentující dialogové okno:

```
...
dialogOkno = $( "#dialogDiv" ).dialog({
  autoOpen: false,
  minWidth: 900,
  maxHeight: 600,
  modal: true
});
...
```

Poté již stačí takto vytvořený objekt naplnit konfiguračními informacemi za pomoci příkazu `$(#dialogDiv").html(informace)` a zobrazit jej příkazem `dialogOkno.dialog("open")`. Výsledné dialogové okno je znázorněno na obrázku 12. Zobrazilo se po kliknutí na možnost „routingTable“. Zobrazené informace se dají kopírovat, neboť se zobrazují jako prostý text, což může být zejména pro správce sítě velmi užitečné. Formátování textu bylo zachováno tak, jak jej provádí zařízení.



Obrázek 12: Ilustrativní příklad dialogového okna

6.5 Objevené problémy při implementaci

Při provádění implementace se vyskytly problémy, které bylo potřeba vyřešit. Některé budou v kapitole představeny. Může to být užitečné pro ty, kteří by se touto problematikou chtěli také zabývat.

6.5.1 ARP tabulka

Při parsování tabulky *ARP* se vyskytly dva problémy, které bylo nutné vyřešit:

1. sloupec „Age“ v *ARP* tabulce,
2. záznam „Incomplete“ namísto MAC adresy.

První problém spočíval v tom, že tabulka obsahuje sloupec „Age“, který značí stáří *ARP* záznamu. Některé záznamy ale měly tento sloupec prázdný (pouze znaky mezer), což při parsování dat dělalo značné potíže, protože takový sloupec nebyl vůbec rozpoznán (pro rozpoznání musí ve sloupci být alespoň jeden neprázdný znak).

Problém byl vyřešen tak, že se přidala podmínka `if`, která se ptala na počet sloupců. Pokud jich bylo méně, než se předpokládá, byl uměle dodán další sloupec na místo, kde by se měla nacházet hodnota „Age“:

```
...
if(tempColumn.size() < 7) tempColumn.add(2,"-");
...
```

Druhý problém nastal v situaci, kdy byl vyslán příkaz „ping“ z důvodu obnovení tabulek na síťových zařízeních v cestě. Pokud zařízení, z něhož byl příkaz poslán, nemělo záznam v tabulce *ARP* pro cílovou IP adresu, nastal problém. Příkaz ping totiž inicializoval *ARP* dotaz. A pokud na něj ještě nepřišla odpověď, zařízení si v tabulce uchová záznam pro danou IP adresu s tím, že ve sloupci pro MAC adresu uloží slovo „Incomplete“.

Problém se vyřešil pouhým jedním řádkem, za pomoci podmínky `if`. Zařízení si totiž pro nedokončený *ARP* dotaz ve sloupci pro rozhraní („Interface“) uchovává pouze bílé znaky (mezery). Tudíž se lze jednoduše zeptat, zdali tento sloupec má nějakou hodnotu. Pokud ne, je zřejmé, že se jedná o nedokončený *ARP* dotaz:

```
...
String secondCol = tempColumn.get(6).trim();
if (!secondCol.tequals("")) secondCol =
   Unifier.unifyNetInterface(secondCol);
...
```

6.5.2 Hostname zařízení

Další problém, který vyplynul až z průběžného testování, souvisí s *CDP* tabulkou. Pokud má zařízení nastaveno doménové jméno příkazem `ip domain-name`, objevuje se toto jméno v *CDP* tabulce za názvem („hostname“) zařízení (tedy např. „SW1.domenoveJmeno“).

Vzhledem k tomu, že databáze u každého síťového zařízení uchovává pouze „hostname“, nešlo by jej porovnávat se získaným názvem z *CDP*, jenž obsahuje doménové jméno.

Problém lze jednoduše vyřešit tak, že se odebere přidané doménové jméno, čímž se získá pouze čistý název („hostname“) zařízení. Tohle řešení implementuje metoda `unifyHostname(...)` ve třídě „Unifier“:

```
...
public static String unifyHostname(String hostname){
  if(hostname.contains(".")) {
    String[] splitted = hostname.split("\\.");
    return splitted[0];
  }
  return hostname;
}
...
```

Řešení má však jedno omezení – v názvu zařízení nelze používat tečky. Nicméně takový znak se většinou v názvu zařízení nevyskytuje.

7 Testování aplikace

K ověření správnosti implementace aplikace je nezbytné ji otestovat. O tom, jak je testování vyvíjené aplikace provedeno, pojednává celá tato kapitola.

Problematika počítačových sítí je velmi rozsáhlá. Kombinací, jak a z čeho je síť složená, je nespočet. Aby se dalo ověřit, jestli by aplikace obstála v reálném prostředí, je třeba vytvořit komplexní sít, která obsahuje principy, jež jsou běžné v takovém prostředí.

K tomu poslouží předložená topologie, která byla sestavena v reálném prostředí za pomoci reálných zařízení. Je vytvořeno více scénářů, reflektujících odlišné situace, jež jsou typické pro reálné prostředí. Scénáře se liší zejména různými kombinacemi zdrojového a cílového bodu, ale také pozměněnou konfigurací – to proto, aby vznikla překážka ve hledání cesty. A protože jedním z cílů aplikace je umět rozpoznat tyto problémy v cestě, lze tímto tuto funkcionalitu otestovat.

7.1 Topologie

Aplikace je testována na komplexní topologii, čítající sedm síťových zařízení a sedm koncových uzlů (PC). Je vyobrazena na obrázku 13. Návrh je udělán tak, aby splňoval předpoklady třívrstvého síťového návrhu[29]. Šest koncových uzlů („PC-1“ až „PC-6“) je zapojeno do přístupových přepínačů („sw-a-1“ až „sw-b-1“) a ty do páteřních L3 přepínačů („core“). Na sedmém počítači („mgmt“) bude spouštěna aplikace. Je tedy zřejmé, že PC musí být součástí sítě, jež zpřístupňuje vzdálený přístup k zařízením.

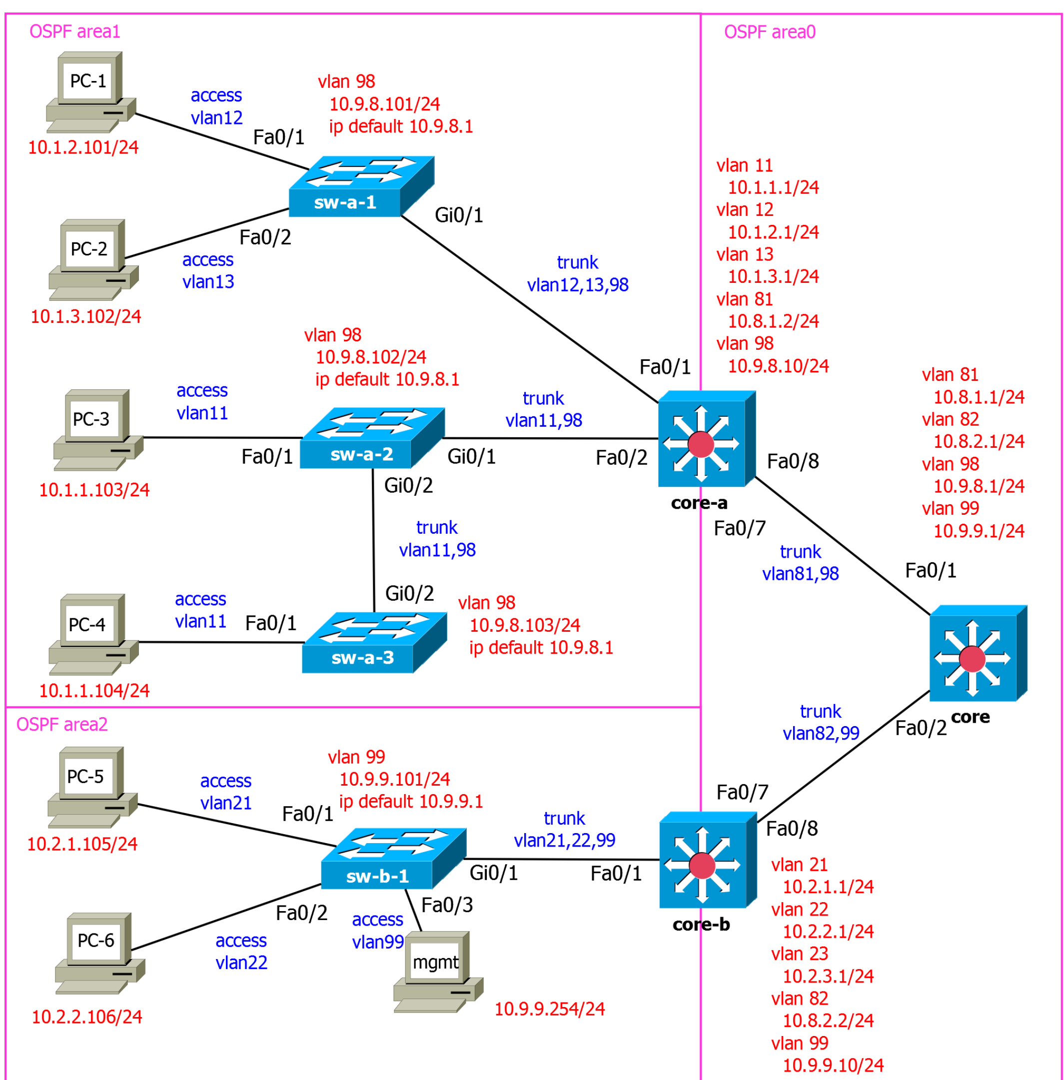
7.2 Popis konfigurace

V topologii existuje deset různých VLAN. Každá z nich má nastavenou IP adresu sítě a IP adresu výchozí brány. Pro přesné nastavení slouží tabulka 1. IP adresa výchozí brány je vždy realizována na určitém L3 zařízení, prostřednictvím virtuálního rozhraní (SVI).

Koncové uzly jsou vždy součástí určité VLAN, tudíž jejich IP adresa se musí respektovat na základě VLAN IP adresy sítě. Přesné nastavení popisuje tabulka 2. Rozhraní (switchporty) síťových zařízení mohou být buď typu *Access* nebo *Trunk*. Na obrázku 13 je to vyznačeno u každého spoje modrou barvou.

Aby aplikace mohla fungovat, musí mít vzdálený přístup ke každému síťovému zařízení. To je realizováno pomocí virtuálních rozhraní (SVI), jež jsou součástí VLAN 98 nebo VLAN 99, které slouží právě pro tento účel. Každé SVI má unikátní IP adresu a přihlašovací údaje. Obojí lze najít v tabulce 3.

[29]Třívrstvý síťový návrh je definován společností Cisco. Jedná se o návrh, který zahrnuje tři vrstvy – přístupovou, distribuční a jádra sítě (Teare, 2003). Koncové uzly jsou připojovány do přístupových přepínačů na přístupové vrstvě. Přístupové přepínače jsou napojeny do páteřní vrstvy, kterou obvykle tvoří rychlé L3 přepínače.



Obrázek 13: Komplexní topologie pro testování

Směrování zajišťuje dynamický směrovací protokol OSPF. Topologie byla rozdělena do tří oblastí („OSPF areas“), které jsou na obrázku 13 znázorněny růžovými obdélníky.

Každé síťové zařízení má svou vlastní konfiguraci. Přihlašovací údaje k zařízením jsou permanentně uložené v *in-memory* databázi ve speciálním souboru. Proto veškeré konfigurace všech zde předkládaných zařízení a souborů, reprezentující ukládaná data v grafové i *in-memory* databázi, jsou součástí přílohy.

Tabulka 1: Nastavení VLAN

Číslo	Pojmenování	IP adresa sítě/maska	IP adresa výchozí brány
11	Budova A - učebny	10.1.1.0/24	10.1.1.1
12	Budova A - kanceláře	10.1.2.0/24	10.1.2.1
13	Budova A - kamery	10.1.3.0/24	10.1.3.1
21	Budova B - učebny	10.2.1.0/24	10.2.1.1
22	Budova B - kanceláře	10.2.2.0/24	10.2.2.1
23	Budova B - kamery	10.2.3.0/24	10.2.3.1
81	Spoj Core-A Core	10.8.1.0/24	10.8.1.1
82	Spoj Core-B Core	10.8.2.0/24	10.8.2.1
98	Management budova A	10.9.8.0/24	10.9.8.1
99	Management	10.9.9.0/24	10.9.9.1

Tabulka 2: Nastavení koncových uzlů – PC

Název	zařazení do VLAN	IP adresa/maska
PC-1	VLAN 12	10.1.2.101/24
PC-2	VLAN 13	10.1.3.102/24
PC-3	VLAN 11	10.1.1.103/24
PC-4	VLAN 11	10.1.1.104/24
PC-5	VLAN 21	10.2.1.105/24
PC-6	VLAN 22	10.2.2.106/24
mgmt	VLAN 99	10.9.9.254/24

Tabulka 3: Přihlašovací údaje k zařízením

Název	Management IP adresa	Přihlašovací jméno	Heslo
sw-a-1	10.9.8.101	admina	ciscoa
sw-a-2	10.9.8.102	admina	ciscoa
sw-a-3	10.9.8.103	admina	ciscoa
sw-b-1	10.9.9.101	admin	cisco
core-a	10.9.8.10	admina	ciscoa
core-b	10.9.9.10	admin	cisco
core	10.9.8.1	admin	cisco

7.3 Scénáře bez problému

Byly provedeny čtyři scénáře s různými zdrojovými a cílovými uzly. Síť je nakonfigurována tak, aby měl každý uzel plnou konektivitu s jakýmkoliv jiným uzlem v topologii.

7.3.1 První scénář

První nalezení cesty je uskutečněno z počítače „PC-1“ na počítač „PC-2“. Jak je vidět na obrázku topologie, oba uzly jsou připojeny ke stejnému přepínači. Nicméně každý je v jiné VLAN, tudíž zde probíhá směrování mezi VLAN na prvku „core-a“, tzv. „inter-VLAN routing“. Scénář má tedy prokázat, že je aplikace schopna si s tímto poradit a správně zobrazit výsledek.

Obrázek 14 dokazuje, že je nalézána správná cesta. Počítač musí jít nejdříve na svoji výchozí bránu („core-a“), která mu sdělí, že se musí vrátit zpět, kde najde cílový uzel.

7.3.2 Druhý scénář

Druhý scénář je uskutečněn z „PC-4“ do „PC-5“. Počítače jsou takto vybrány záměrně proto, aby byly otestovány následující věci:

- průchod přes více L2 přepínačů za sebou,
- směrování přes vícero L3 zařízení.

Na obrázku 15 je vidět, že byla cesta nalezena správně. Vede přes všechny tři OSPF oblasti, takže je zřejmé, že se směrováním si aplikace ví rady.

7.3.3 Třetí scénář

Ve třetím scénáři je ověřována komunikace mezi „PC-3“ a „PC-4“. Zařízení jsou ve stejné síti, tudíž by komunikace měla zůstat na úrovni L2 zařízení. Tímto je ověřováno, zdali aplikace umí zjistit, že se jedná o prvky v jedné síti, a tedy není potřeba pakety směrovat.

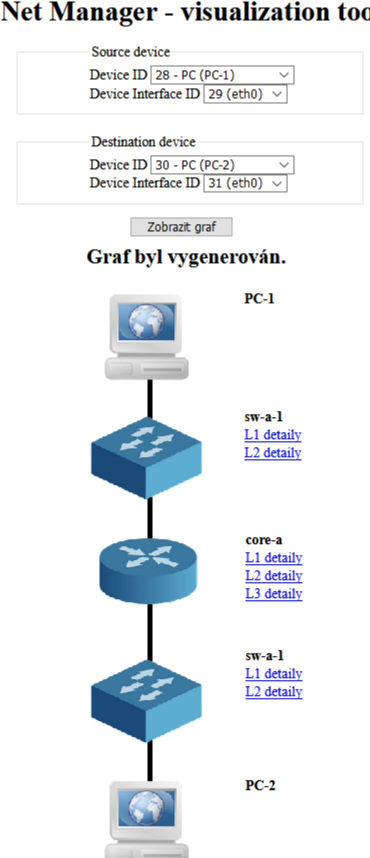
Obrázek 16 ukazuje výsledek hledání cesty. Dokazuje, že je komunikace uskutečňována pouze v rámci L2 přepínačů.

7.3.4 Čtvrtý scénář

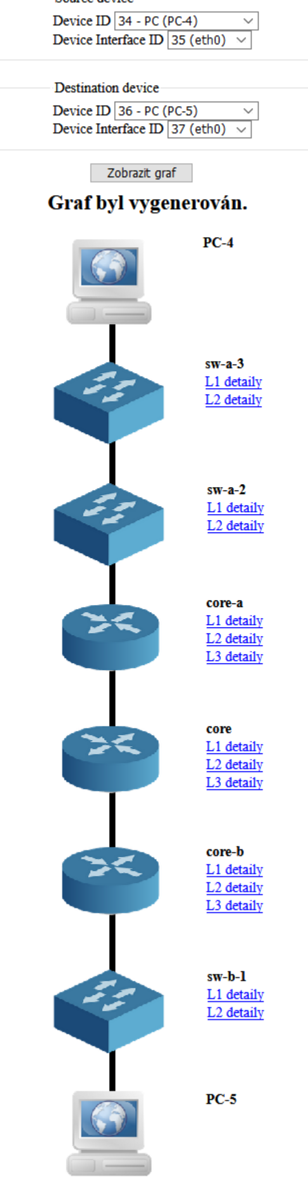
Čtvrtý scénář testuje komunikaci z „PC-1“ do „PC-6“. Navíc byla provedena změna v konfiguraci. Mezi prvkem „core-a“ a „core-b“ byl přidán spoj, který je stejně rychlý a funkční jako všechny ostatní. Tím byla vytvořena alternativní cesta, která by měla být (ve výchozí konfiguraci) rychlejší. Je tím testováno, zdali se aplikace správně rozhodne jít tímto spojem, ačkoliv má na výběr více alternativ.

Na obrázku 17 je vidět, že se aplikace rozhoduje správně. Místo aby šla cestou, kterou zvolila v kapitole 7.3.2, vybrala správně cestu novou.

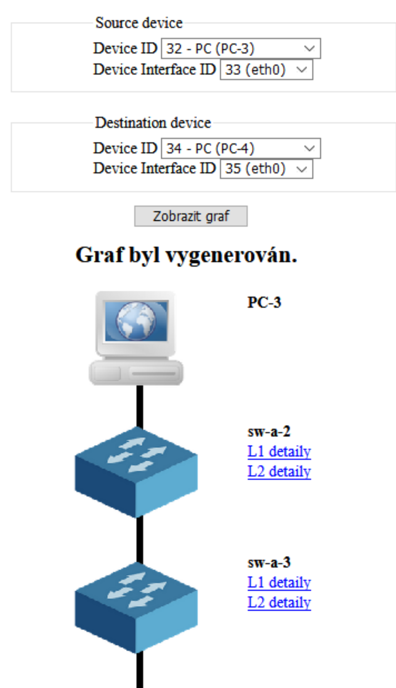
Že funguje příkaz ping, který se uskutečňuje pro aktualizaci tabulek na zařízeních (viz kapitola 5.4.3), dokazuje obrázek 18. Je v něm vidět odchyt dat v programu *Wireshark*. Z jednotlivých řádků lze vyčíst, že probíhala komunikace (ping) mezi výchozí bránou „core-a“ a koncovým uzlem „PC-1“.



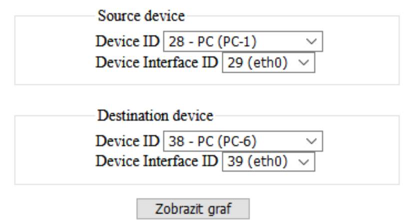
Obrázek 14: Výsledek prvního bezproblémového scénáře

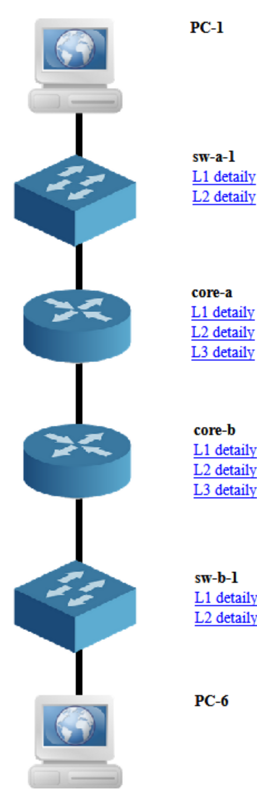


Obrázek 15: Výsledek druhého bezproblémového scénáře



Obrázek 16: Výsledek třetího bezproblémového scénáře





Obrázek 17: Výsledek čtvrtého bezproblémového scénáře

*Ethernet 2

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

icmp

No.	Time	Source	Destination	Protocol	Length	Info
36	18…	10.1.2.1	10.1.2.101	ICMP	114	Echo (ping) request id=0x0033, seq=0/0, ttl=255 (reply in 37)
37	18…	10.1.2.101	10.1.2.1	ICMP	114	Echo (ping) reply id=0x0033, seq=0/0, ttl=128 (request in 36)
38	18…	10.1.2.1	10.1.2.101	ICMP	114	Echo (ping) request id=0x0033, seq=1/256, ttl=255 (reply in 39)
39	18…	10.1.2.101	10.1.2.1	ICMP	114	Echo (ping) reply id=0x0033, seq=1/256, ttl=128 (request in 38)

Obrázek 18: Ukázka příkazu ping pro obnovu tabulek

7.4 Scénáře s problémem

Ke každému scénáři bez problému byl uskutečněn další scénář, ve kterém se změnila konfigurace záměrně tak, aby nastal v síti problém. Zdrojové a cílový uzly zůstaly nezměněny. Cílem je ukázat, jak vypadá aplikace v případě, že nelze najít kompletní cestu.

Kompletní sadu příkazů, nutných pro zásahy do konfigurace, lze najít v elektronické příloze. Jsou tam také vloženy zajímavé výpisy tabulek, které mohou přiblížit konkrétní chování zařízení při změně konfigurace.

7.4.1 První scénář

V prvním scénáři je provedena změna konfigurace na prvku „sw-a-1“, konkrétně na rozhraní „Fa0/2“, které bylo pomocí příkazu `shutdown` (v konfiguračním módu pro rozhraní) vypnuto. Tím se znemožní komunikace mezi „PC-2“ a „sw-a-1“. Kompletní cesta by tedy neměla být nalezena.

Obrázek 19 potvrzuje, že to aplikace udělá opravdu tak, jak se předpokládalo. Navíc uživatele informuje o pravděpodobné chybě. V té je řečeno, že L3 zařízení „core-a“ nemůže najít záznam pro cílové zařízení v ARP tabulce, což je pravda.

7.4.2 Druhý scénář

Ve druhém scénáři je podniknut razantnější zásah. Vypne se prvek „core-a“, který v tomto případě slouží jako výchozí brána pro zdrojový uzel „PC-4“. Vzhledem k tomu, že jsou koncové prvky v jiné síti, je potřeba výchozí bránu využít ihned na začátku hledání cesty pro získání první cílové MAC adresy.

Na obrázku 20 je vidět, že se aplikaci skutečně nedaří spojit s výchozí bránou na IP adrese „10.9.8.10“, což je rozhraní na prvku „core-a“ pro vzdálený přístup. Jako důvod uvádí vypršení doby pro připojení.

7.4.3 Třetí scénář

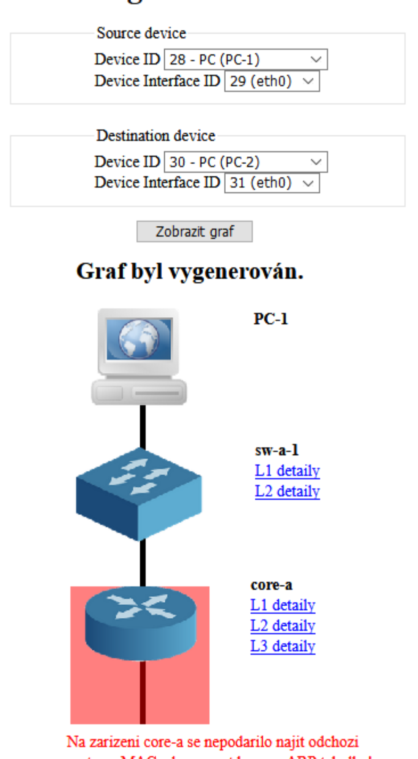
Ve třetím scénáři se změní konfigurace na zařízení „sw-a-2“, konkrétně na rozhraní „Gi0/2“. Příkazem `switchport trunk allowed vlan remove 11` (v konfiguračním módu pro rozhraní) se zakáže propouštět pakety s označením pro VLAN 11, což má za následek neúspěšnou komunikaci mezi koncovými uzly, neboť jsou oba ve VLAN 11.

Obrázku 21 dokládá, že přepínač „sw-a-2“ neví, jak pokračovat dále, protože mu pro cílovou MAC adresu chybí záznam v CAM tabulce. Ovšem cílová MAC adresa známá je, neboť výchozí brána si v ARP tabulce udržuje záznam pro cílovou IP adresu, protože ještě nevypršela jeho platnost.

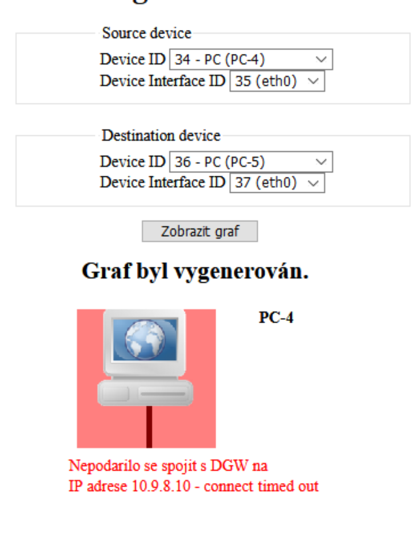
7.4.4 Čtvrtý scénář

Poslední čtvrtý scénář má za úkol nabourat správné směrování. Konfigurace a zapojení je (oproti čtvrtému scénáři bez chyby) opět výchozí. Tudíž neexistuje přímé spojení mezi „core-a“ a „core-b“. Na zařízení „core-b“ je, v rámci směrovacího protokolu OSPF, vypnuta distribuce sítí pro VLAN 21 a VLAN 22. Uskuteční to příkaz `no network 10.2.0.0 0.0.255.255 area 2` v konfiguračním módu pro směrovací protokol OSPF.

Z obrázku 22 lze vyčíst, že byla komunikace neúspěšná už od prvku „core-a“. To proto, že se mu odstranily záznamy ve směrovací tabulce pro cílovou IP adresu počítače „PC-6“. Chyba, jež je vypsána, dokazuje, že šlo opravdu o tento typ problému (nenalezení záznamu ve směrovací tabulce).

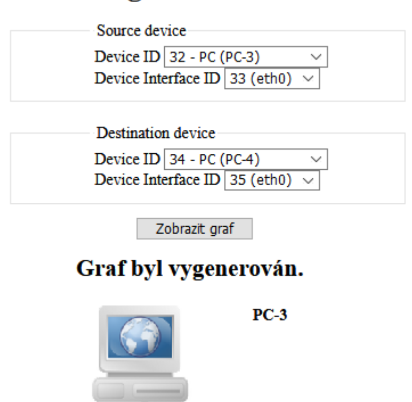


Obrázek 19: Výsledek prvního scénáře s problémem



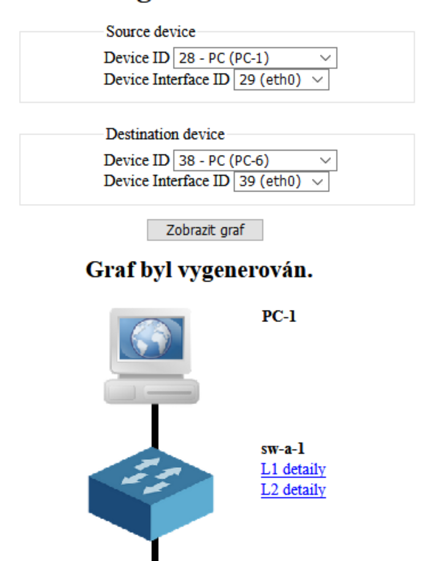
Obrázek 20: Výsledek druhého scénáře s problémem

Net Manager - visualization tool



Obrázek 21: Výsledek třetího scénáře s problémem

Net Manager - visualization tool



Obrázek 22: Výsledek čtvrtého scénáře s problémem

8 Diskuse a závěr

Cílem této práce bylo vytvořit aplikaci pro vizualizaci komunikačního procesu v počítačové síti. Ta má uživateli zobrazit postupně všechny prvky, které se nacházejí v cestě z určeného zdrojového bodu k bodu cílovému.

Bylo nutné provést analýzu současného stavu existujících aplikací, které se zabývají touto problematikou. To proto, aby se zjistilo, zdali již na zadaný problém neexistuje řešení. Nalezené aplikace byly srovnávány s přesně vytyčenými parametry. Hledalo se jak v závěrečných pracích, tak na Internetu, prostřednictvím vyhledávače na základě klíčových slov. Závěrem celé analýzy bylo, že neexistuje aplikace, která by splňovala všechny definované parametry, tudíž je zde prostor pro tvorbu nové aplikace. Zbytek práce již pojednává tomto novém řešení, které se snaží splnit všechny definované body a přinést tuto novou funkcionalitu.

Vůbec první podnět pro tvorbu takové aplikace zazněl z řad pracovníků Mendelovy univerzity v Brně, proto byla práce usměrňována tak, aby mohla univerzitě v budoucnosti posloužit. Existuje totiž projekt, který má přinést novou aplikaci, starající se o správu univerzitní počítačové sítě. Tudíž i vyvíjená aplikace byla navržena a implementována tak, aby mohla být v budoucnu součástí tohoto projektu, což bylo i jedním z cílů, který se podařilo naplnit.

Návrh aplikace měl dvě části. První část popisovala vztah k existující aplikaci. Byly v ní navrženy změny, které jsou nutným předpokladem pro správné fungování nově vyvíjené části. Druhá část se zabývala již návrhem nového řešení. Byl předložen návrh jak backendové části aplikace, která nalézá prvky v cestě, tak frontendové části, která zobrazí uživateli výsledky hledání.

V implementační části byly probrány nejdůležitější funkční části aplikace. Bylo předvedeno například to, jak se provádí „parsování“ dat, nebo jak je implementováno připojování se k prvkům.

Poslední kapitola se zabývala testováním aplikace. Měla prokázat, že implementované řešení obstojí při použití v reálném prostředí. Byla vytvořena komplexní počítačová síť, složená ze sedmi reálných síťových zařízení a sedmi koncových uzlů (počítačů). Osm odlišných scénářů na této síti mělo za úkol otestovat aplikaci v různých situacích. Čtyři z nich testovaly úspěšnou komunikaci, ve zbylých byla změněna konfigurace sítě tak, aby nastal nějaký problém, který by měla aplikace odhalit. Výsledek testování ukázal, že aplikace se v různých situacích chovala tak, jak bylo předpokládáno a chtěno.

Hlavní cíl aplikace se podařilo splnit. Backendová část existujícího projektu byla obohacena o kód, který se stará o nalezení cesty. Bohužel se nepodařilo implementovat vizualizaci nalezené sítě do existujícího frontendu, psaném ve frameworku *AngularJS*, protože práce a zjištěných problémů na backendové části bylo více, než se předpokládalo. Nicméně byl vytvořen vlastní frontend, který vizualizuje nalezený výsledek uživateli a umožňuje navíc také nahlédnout do konfigurace každého nalezeného zařízení. Dílčím cílem práce bylo takový výpis uživateli nabídnout, tudíž i ten byl splněn.

Aplikace měla být uživatelsky přívětivá. Proto vytvořený frontend, mající za úkol interakci s uživatelem, nebyl navržen nijak složitě. Uživatel musí pouze zvolit zdrojový a cílový prvek, resp. rozhraní, čímž definuje počáteční a koncový bod hledání. Poté už jen stačí, aby se stisknutím tlačítka spustil celý proces nalezení cesty, který posléze vygeneruje graf, reprezentující výsledek.

8.1 Omezení a předpoklady

Předkládané řešení má jistá omezení a předpoklady, která musejí být respektována a dodržena pro správné fungování.

Jedním z nejpodstatnějších omezení je, že aplikace předpokládá použití síťových zařízení pouze od firmy Cisco. S prvky jiných společností neumí pracovat. Důvod je ten, že prvky od různých společností se liší příkazy v příkazové řádce. Je tedy zřejmé, že pokud by byla počítačová síť složená z prvků od různých výrobců, aplikace by nefungovala správně. Bohužel i u některých operačních systémů produktů Cisco se používají rozdílné příkazy, tudíž nelze zaručit stoprocentní funkčnost ani zde.

Pokud se dodrží předpoklad síťových zařízení pouze od firmy „Cisco“, je třeba splnit ještě jeden. Každé zařízení musí podporovat a mít zapnutý proprietární protokol *CDP*. Na základě tohoto protokolu se zjišťují přímo připojená zařízení (sousední prvky). Pokud by byl protokol nedostupný, aplikace skončí s chybovou hláškou, oznamující nemožnost nalezení sousedního prvku. V produkci se bohužel často stává, že se protokol záměrně vypíná, zejména z důvodu bezpečnosti. Pokud má být aplikace použitelná, je nezbytné s tímto předpokladem počítat.

Dalšími omezeními pro správnou funkčnost jsou:

- každé zařízení má nastaveno přihlašovací jméno s úrovní oprávnění 15,
- nejsou použity určité pokročilé technologie (např. *VRF*, *ACL*, port-security),
- počítače mají nastavenou výchozí bránu tak, jak je to definováno v databázi.

Pokud by zařízení nemělo přihlašovací jméno s úrovní oprávnění 15, nepodařilo by se s ním navázat spojení, a aplikace by tedy označila tento prvek za chybový.

Pokud by se v konfiguraci prvků objevily pokročilé technologie, jako je např. *VRF* (virtuální instance směrovací tabulky), aplikace by si s nimi neuměla poradit, a tudíž by skončila s prohledáváním. To může být určitým omezením v reálném prostředí, protože se nevylučuje použití nepodporovaných protokolů a technologií.

Zcela jasné omezení vyplývá z podstaty návrhu a implementace aplikace. Počítá se totiž s tím, že pro každou síť existuje unikátní grafová databáze, naplněná takovými daty, aby reflektovala skutečný stav sítě (viz kapitola 5.4.2). Jestliže by tedy chtěl uživatel vyhledat cestu mimo spravovanou síť, nelze mu vyhovět.

Pokud by určitý počítač neměl správně nastavenou výchozí bránu tak, jak je to určeno v databázi, mohl by nastat problém. Cesta by sice mohla být nalezena, nicméně ve skutečnosti by počítač neměl konektivitu, jak by se z výsledku mohlo zdát.

8.2 Možné návrhy na vylepšení aplikace do budoucna

Asi nejpodstatnějším možným vylepšením (z hlediska přínosu aplikace) do budoucna se jeví nezávislost na platformě síťových zařízeních. Bylo by tedy potřeba do řešení zařadit modul, který by se staral o zjištění výrobce zařízení a na základě toho by využil další modul, který by přinášel konkrétní příkazy a funkcionalitu pro zařízení daného výrobce.

Dále by se dalo polemizovat o tom, jak vyřešit vizualizaci cesty, která sahá za hranice „známé“ sítě. V databázi nelze mít uložená data, která by popisovala skutečný stav Internetu. Možnou variantou je použít technologii *Traceroute*, která umožňuje zjistit L3 zařízení v cestě. Mohlo by to fungovat tak, že pokud by uživatel chtěl jít ze známé částí sítě do neznámé, dokázala by aplikace zobrazit kompletní cestu ve známé síti. Poté by zobrazila jen ta L3 zařízení (bez jejich konfigurace, pouze IP adresy), která by nalezl zmíněný *Traceroute*. Nevýhodou je, že mnohdy je tato technologie na zařízeních nedostupná.

Aplikace by mohla uživateli umožňovat vybrání zdrojového a cílového bodu i za použití jiných kritérií (např. pomocí doménových jmen, IP adres atp.). Vzhledem k tomu, jak je aplikace navržena a implementována, to není žádný problém. Jde jen o to, stanovit si další možnou variantu a implementovat ji do existujícího repertoáru nabízených způsobů.

Vzhledem k tomu, že byl vytvořen vlastní frontend, který není součástí existujícího řešení, nabízí se zde možnost implementovat vizualizaci výsledku hledání cesty do tohoto existujícího řešení.

9 Reference

AJAX Introduction, *w3schools.com* [online]. 2017 [cit. 2017-27-04]. Dostupné z: https://www.w3schools.com/xml/ajax_intro.asp.

Apache Commons Net, *Apache Commons* [online]. 2017 [cit. 2017-26-04]. Dostupné z: https://commons.apache.org/proper/commons-net/.

BOUŠKA, Petr. *SNMP: Simple Network Management Protocol* [online]. 2006 [cit. 2017-04-07]. Dostupné z: http://www.samuraj-cz.com/clanek/snmp-simple-network-management-protocol.

CytoScape [online]. c2001-2016 [cit. 2016-11-20]. Dostupné z: http://www.cytoscape.org/.

ČÁPKA, David. Úvod do jQuery, *ITnetwork.cz* [online]. 2017 [cit. 2017-26-04]. Dostupné z: https://www.itnetwork.cz/javascript/jquery-zaklady/javascript-tutorial-funkcionalni-programovani-a-jquery-webova-kalkulacka.

Dialog, *jQuery user unterface* [online]. 2017 [cit. 2017-17-05]. Dostupné z: https://jqueryui.com/dialog/.

DOSTÁLEK, L., KABELOVÁ, A. *Velký průvodce protokoly TCP/IP a systémem DNS*. 3. vyd. Brno: Vydavatelství a nakladatelství CP Books, a.s., 2005. ISBN 80-7226-675-6.

HEJNA, Martin. *Vizualizace stavu sítě*. Praha, 2010. Diplomová práce. Vysoká škola ekonomická v Praze, Fakulta informatiky a statistiky, Katedra informačních technologií. Vedoucí práce Luboš Pavlíček.

Intro to Cypher, *Neo4J* [online]. 2017 [cit. 2017-26-04]. Dostupné z: https://neo4j.com/developer/cypher-query-language/.

jQuery, *Wikipedie* [online]. 2017 [cit. 2017-26-04]. Dostupné z: https://cs.wikipedia.org/wiki/JQuery.

jQuery ajax() Method, *w3schools.com* [online]. 2017 [cit. 2017-27-04]. Dostupné z: https://www.w3schools.com/jquery/ajax_ajax.asp.

KRČMÁŘ, Petr. Bezpečné přihlašování na server pomocí SSH klíče, *Coolhousing.net* [online]. 2016 [cit. 2017-04-25].Dostupné z: https://www.root.cz/clanky/jak-se-prihlasovat-na-ssh-bez-zadavani-hesla.

KUROSE, James F. *Počítačové sítě*. 1. vyd. Brno: Computer Press, 2014. 622 s. ISBN 978-80-251-3825-0.

LiveAction [online]. Palo Alto, c2016 [cit. 2016-11-20]. Dostupné z: http://www.liveaction.com/.

MALÝ, MARTIN. LESS: stejné CSS za méně peněz, *zdroják.cz* [online]. 2010 [cit. 2017-14-05]. Dostupné z: https://www.zdrojak.cz/clanky/less-stejne-css-za-mene-penez/.

MALÝ, MARTIN. REST: architektura pro webové API, *zdroják.cz* [online]. 2009 [cit. 2017-29-04]. Dostupné z: https://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/.

MATOUŠEK, MARTIN. *Vizualizace síťového provozu*. Brno, 2014. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Michal Kováčik.

NetDepict [online]. c2010-2012 [cit. 2016-11-20]. Dostupné z: http://www.netdepict.net/.

Network Topology Mapper: Automatically map your network in minutes. *SolarWinds: The Power of Manage IT* [online]. c2003-2016 [cit. 2016-11-20]. Dostupné z: http://www.solarwinds.com/network-topology-mapper.

Network Troubleshooting: Definition - What does Network Troubleshooting mean. *Techopedia* [online]. c2016 [cit. 2016-11-20]. Dostupné z: https://www.techopedia.com/definition/29989/network-troubleshooting.

ODOM, WENDELL. *Cisco CCNA routing and switching ICDN2 200-101: official cert guide*. Indianapolis: Cisco Press, 2013. 717 s. ISBN 978-1-58714-373-1.

PÁNIK, MILAN. *Vizualizácia sieťových topologií*. Brno, 2014. Bakalářská práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce Petr Holub.

ROUSE, MARGARET. In-memory database, *WhatIs.com* [online]. 2012 [cit. 2017-28-04]. Dostupné z: http://whatis.techtarget.com/definition/in-memory-database.

Spring Data JPA - Reference Documentation, *Spring.io* [online]. 2016 [cit. 2017-29-04]. Dostupné z: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/.

SSH – bezpečné používání vzdáleného počítače a kopírování dat, *dsl.cz* [online]. 2017 [cit. 2017-04-25]. Dostupné z: http://www.dsl.cz/jak-na-to/jak-na-ssh.

ŠTĚPÁNEK, MAREK. Jak na vzdálený přístup?, *ICTSecurity* [online]. 2009 [cit. 2017-04-25]. Dostupné z: http://www.ictsecurity.cz/odborne-clanky/2153-jak-na-vzdaleny.

TAUER, Tomáš. *Uživatelské rozhraní pro vizualizaci rozsáhlých sítí*. Brno, 2013. Bakalářská práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce David Svoboda.

TEARE, Diane. *Návrh a realizace sítí Cisco*. 1. vyd. Brno: Vydavatelství a nakladatelství Computer Press, 2003. ISBN 80-251-0022-7.

Telnet, *Správa sítě – slovník pojmů* [online]. 2016 [cit. 2017-04-25]. Dostupné z: http://www.sprava-site.eu/telnet/.

The Network Visualizer [online]. 2008 [cit. 2016-11-20]. Dostupné z: http://tnv.sourceforge.net/.

Vysokoškolské kvalifikační práce [online]. Fakulta informatiky Masarykovy univerzity, 2016 [cit. 2016-11-20]. Dostupné z: http://theses.cz/.

What is a Graph Database, *Neo4J* [online]. 2017 [cit. 2017-26-04]. Dostupné z: https://neo4j.com/developer/graph-database/.

What is JAVA?, *JAVA* [online]. 2017 [cit. 2017-26-04]. Dostupné z: http://searchmicroservices.techtarget.com/definition/Java.

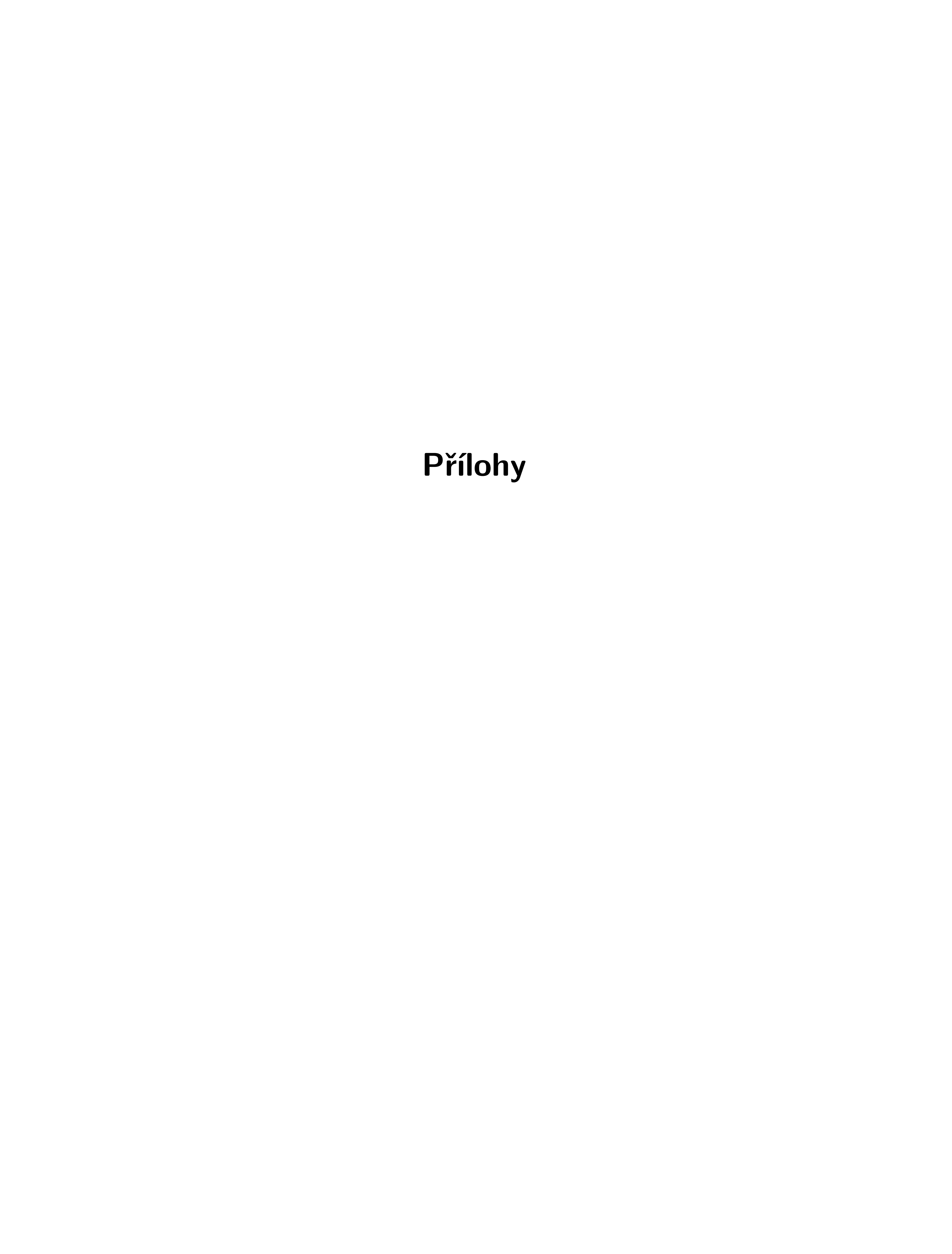
Závěrečné práce. *Informační systém Masarykovy univerzity* [online]. 2016 [cit. 2016-11-20]. Dostupné z: http://is.muni.cz/thesis/.

Závěrečné práce. *Mendelova univerzita v Brně* [online]. 2016 [cit. 2016-11-20]. Dostupné z: https://is.mendelu.cz/zp/?lang=cz.

Závěrečné práce. *Vysoké učení technické v Brně* [online]. Brno, 2016 [cit. 2016-11-20]. Dostupné z: https://www.vutbr.cz/studium/zaverecne-prace.

10SCAPE [online]. Canada, 2016 [cit. 2016-11-20]. Dostupné z: https://10scape.com/.

10-Strike Software [online]. Russian Federation, c1998-2016 [cit. 2016-11-20]. Dostupné z: http://www.10-strike.com/lanstate/.



Přílohy

A Seznam zkratek

ACL Access Control List

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

ARP Address Resolution Protocol

CAM Content Addressable Memory

CDP Cisco Discovery Protocol

CPU Central Processing Unit

CSS Cascading Style Sheets

FTP File Transfer Protocol

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

ICMP Internet Control Message Protocol

IP Internet Protocol

JSON JavaScript Object Notation

LESS Leaner CSS

MAC Media Access Control

OSPF Open Shortest Path First

PC Personal Computer

RAM Random Access Memory

SMTP Simple Mail Transfer Protocol

SNMP Simple Network Management Protocol

SQL Structured Query Language

SSH Secure Shell

UML Unified Modeling Language

URL Uniform Resource Locator

VLAN Virtual LAN

VRF Virtual Routing and Forwarding

B Elektronická příloha

Elektronická příloha obsahuje:

- kompletní zdrojové kódy – složka „Zdrojové kódy“,
- konfigurace použitých zařízení v sítích – složka „Konfigurace zařízení“,
- změny v konfiguracích jednotlivých zařízení pro účely testování – složka „Konfigurace zařízení\Komplexní topologie“ – adresáře, nesoucí názvy zařízení,
- konfigurace databází – složka „Konfigurace databází“,
- kompletní UML diagramy – složka „Kompletní UML diagramy“.