České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Rozšíření grafického uživatelského rozhraní NetopeerGUI

Bc. David Alexa

Vedoucí práce: Ing. Tomáš Čejka

27. dubna 2015

Poděkování

Chtěl bych poděkovat organizaci CESNET za možnost být součástí tohoto projektu a hlavně vedoucímu práce za pomoc při řešení mnohdy nelehkých problémů při vývoji a psaní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 27. dubna 2015 .....

Odkaz na tuto práci

Alexa, David. *Rozšíření grafického uživatelského rozhraní NetopeerGUI.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato práce se zabývá vývojem a rozšířením systému *NetopeerGUI* — webového grafického uživatelského rozhraní pro správu a konfiguraci síťových zařízení. *NetopeerGUI* umožňuje komunikovat s libovolným zařízením podporujícím protokol NETCONF. Jedná se o open–source NETCONF klienta, zveřejněného na serveru GitHub.

NetopeerGUI přináší jednoduchý a intuitivní způsob konfigurace zařízení. Zastiňuje interní příkazy protokolu NETCONF do uživatelsky přívětivé podoby. Usnadňuje konfiguraci díky intuitivnímu uživatelskému rozhraní. Nabízí rozšířené možnosti konfigurace zařízení a modulární rozšíření celé aplikace.

Systém *NetopeerGUI* je implementován v jazyce PHP ve frameworku Symfony2 a využívá webový server *Apache* s modulem *mod_netconf*. Systém byl otestován několika úrovněmi testů. Chování aplikace bylo také podrobeno uživatelskému testování.

Klíčová slova NetopeerGUI, NETCONF klient, Open–source, YANG, PHP, Symfony2, XML, Symfony2 Bundles, jQuery, AJAX, WebSockets, HTML5, PHPUnit, Codeception, SeleniumHQ

Abstract

This thesis deals with development and extension of *NetopeerGUI* — web graphical user interface for network devices management and configuration. *NetopeerGUI* can communicate with any device which supports NETCONF protocol. *NetopeerGUI* is an open–source NETCONF client, published on GitHub.

NetopeerGUI comes with an easy and intuitive way of device configuration. It hides the internal commands of NETCONF protocol and transforms the configuration into a user friendly look. It simplifies the configuration thanks to an intuitive user interface. It comes with extended possibilities of device configuration and with modularity of the whole application.

System *NetopeerGUI* is implemented in PHP using Symfony2 framework and communicates with Apache web server with *mod_netconf* module. The system was tested by different test levels. One of them was user testing of app behavior.

Keywords NetopeerGUI, NETCONF client, Open–source, YANG, PHP, Symfony2, XML, Symfony2 Bundles, jQuery, AJAX, WebSockets, HTML5, PHPUnit, Codeception, SeleniumHQ

Obsah

Seznam obrázků

Úvod

Známe mnoho způsobů, jak nastavit vlastnosti nebo ovlivnit chování různých systémů, elektronické systémy nevyjímaje. Umíme si přestavit nepřebernou škálu přístupů od manuálních přepínačů na analogových rádiích, přes jednoduché konfigurační soubory, výrobcem definované příkazy příkazové řádky až po komplexní nastavení systému v podobě správy operačního systému. Způsobů, jak docílit předem definovaného chování, může být hned několik. Mnohdy je možné kombinovat např. konfiguraci pomocí příkazové řádky či konfiguračními soubory s uživatelsky přívětivějším grafickým uživatelským rozhraním (GUI).

Konfigurace síťových zařízení, kterou se budu zabývat v této práci, probíhá u složitějších systémů často z prostředí příkazové řádky (např. konfigurace směrovacích tabulek v unixových systémech). Na druhé straně pak mohou stát zařízení pro koncové uživatele. V dnešní době mají prakticky všechna zařízení pro koncové uživatele vytvořené jednoduché uživatelské rozhraní. Typickým příkladem může být GUI některého z domácích směrovačů. Díky němu dokáže i méně znalý uživatel nastavit základní parametry své domácí sítě (i za cenu nahlédnutí do manuálu). To vše v prostředí, které obsahuje jemu důvěrně známé ovládací prvky a usnadňuje mu konfiguraci daného systému. Vhodně navržené uživatelské rozhraní dovede mnohdy usnadnit práci také profesionálům, pokud jim např. nabídne rychlejší a efektivnější způsob správy či konfigurace.

Právě na vhodně navržené grafické uživatelské rozhraní nejen pro běžné uživatele, ale i pro profesionály, se zaměřím v této práci. Běžným uživatelům je nutné samotnou konfiguraci libovolného systému co možná nejvíce zjednodušit. Pro mnohé z nich je totiž příliš složité znát, jak fungují jednotlivé procesy uvnitř systému a jakým způsobem jej mají nakonfigurovat. GUI je odstiňuje od manuálních příkazů a složité konfigurace. Znalým profesionálům, pro které je základní způsob konfigurace pomocí příkazové řádky příliš zdlouhavý, dokáže ušetřit mnoho času při ručních úpravách konfigurace.

Představení systému

NetopeerGUI je webová aplikace, jejímž cílem je usnadnit správu a konfiguraci zařízení, které komunikují pomocí protokolu NETCONF [1] (podrobně popsáno v sekci 1.1.1). Velký důraz je kladen na jednoduchost, přehlednost a přívětivost uživatelského rozhraní. Kompletní výčet funkcí aplikace je uveden v sekci 1.5.

Jednou z hlavních předností *NetopeerGUI* je prezentace konfiguračních informací v uživatelsky přívětivé podobě. Oprostíme se tak od nutnosti manuálního psaní mnohdy rozsáhlých XML konfigurací a bez nutné znalosti jednotlivých příkazů protokolu NETCONF. GUI dokáže nahradit často dokonce čtení dokumentace pro získání konkrétních příkazů, dostupných vlastností a hodnot určených pro konfiguraci jednoduchého zařízení jako je např. *toaster* (ukázková implementace *toasteru* je uvedena na stránkách NetconfCentral[1]).

NetopeerGUI je vyvíjeno pod otevřenou licencí a zdrojový kód je veřejně dostupný na serveru GitHub.com[2]. Testovací ukázková instalace *NetopeerGUI* je veřejně dostupná v podobě obrazu virtuálního stroje[3], ve kterém je nainstalováno *NetopeerGUI* včetně všech potřebných nástrojů a knihoven, které je plně funkční i lokálně bez připojení k internetu (demo systém obsahuje také samo sebe jako zařízení, ke kterému je možné se připojit).

Stanovení cílů DP

Cílem mé diplomové práce je rozšíření stávajícího grafického webového uživatelského rozhraní *NetopeerGUI* — klienta pro správu a konfiguraci síťových zařízení. Na *NetopeerGUI* jsem začal pracovat již ve své bakalářské práci [2]. Pro diplomovou práci byly cíle stanoveny na základě reálných požadavků a potřeb uživatelů z praxe.

NetopeerGUI komunikuje se síťovými zařízeními přes protokol NETCONF (více v sekci Protokol NETCONF). Jednou z hlavních výhod *NetopeerGUI* je nezávislost na konfigurovaném zařízení, stačí pouze, aby zařízení umělo komunikovat přes protokol NETCONF. *NetopeerGUI* je webová aplikace, která se připojí k zařízení a umožní tak jeho správu.

Výsledkem této diplomové práce bude vylepšená webová aplikace (dále v textu označována jako *v2.0*), pomocí které bude možné spravovat jakékoliv zařízení používající protokol NETCONF. Nová verze *NetopeerGUI* bude oproti verzi z bakalářské práce (dále v textu označována jako *v1.0*) modulárně i vzhledově rozšiřitelná tak, aby ji mohl používat jakýkoliv uživatel či výrobce podle svých potřeb. Zaměřím se na celkový vzhled a chování aplikace tak,

[1] `http://www.netconfcentral.org/modulereport/toaster`

[2] `https://github.com/CESNET/Netopeer-GUI`

[3] Odkaz pro stažení obrazu virtuálního stroje naleznete na stránkách `https://github.com/CESNET/Netopeer-GUI`

aby byla konfigurace zařízení intuitivní a uživatelsky přívětivá. Doplním také nové, v původní verzi neimplementované, funkčnosti protokolu NETCONF. Rozšířena bude i práce s XML stromem, která v stávající verzi chybí nebo jejíž chování je nedostačující. Aplikace *v2.0* musí umožnit uživateli vytvořit jakoukoliv kompletní konfiguraci zařízení — pomocí implementace všech CRUD operací.

Přehled požadavků

Při sestavování požadavků byl kladen důraz hlavně na zpětnou vazbu, náměty a stížnosti uživatelů, kteří aplikaci *v1.0* používali. Pokrytí funkčnosti NETCONF protokolu a možnosti samotné konfigurace byly v aplikaci *v1.0* do značné míry nedostatečné. V některých, méně častých případech použití, obsahovala aplikace různé druhy chyb. Na vině byla často pouze částečná nebo nedokonale implementovaná funkcionalita (která ale odpovídala původním požadavkům aplikace *v1.0*).

Aplikace *v1.0* (dokončená odevzdáním bakalářské práce) podle původního zadání umožňuje provádět:

- základní operace nad konfiguračními daty ve formě XML stromu (úprava, duplikování) bez nutnosti ručního použití protokolu NETCONF,
- připojit se k více zařízením najednou,
- validace uživatelského vstupu,
- zobrazit konfigurační a stavové informace,
- rozdělit přijaté stavové a konfigurační informace do sekcí s využitím modelů YANG,
- zobrazit historii zařízení, ke kterým se uživatel připojil.

Oproti tomu v požadavcích na diplomovou práci si kladu za cíl odstranit nalezené nedostatky, které původní verze obsahuje a na základě zpětné vazby uživatelů přinést různá vylepšení a rozšíření zejména o:

- plný ajaxový průchod aplikací,
- zobrazení asynchronních notifikací NETCONF serveru,
- modulární rozšíření — každý modul může mít svou vlastní formu výstupu, autor rozšíření si může upravit vzhled celé aplikace,
- rozšíření možností konfigurace o vkládání nových elementů, uživatelské řazení, validaci, volání RPC metod,
- záloha konfigurace zařízení,

- podpora více datových úložišť, včetně kopírování konfigurací mezi nimi,
- jednodušší konfigurace zařízení pomocí předpřipraveného virtuálního obrazu,
- režim jednoho zařízení,
- přihlášení uživatelů pomocí SAML standardu,
- revize zdrojových kódů (např. důkladná práce se jmennými prostory — *XML namespaces*, požadavek vyplývající z NETCONF [1] a YANG [3]),
- zvýšení stability a důkladnější testování.

Zadavatelem této práce je organizace CESNET, z. s. p. o. (dále CESNET). Původní aplikace *v1.0* byla v CESNETu vyvíjena, nasazena a testována. Na základě zpětné vazby od uživatelů vzniklo zadání mé diplomové práce.

Struktura práce

Má diplomová práce je rozdělena do tří základních částí: analýza, realizace a testování. V úvodu jsem Vám krátce představil aplikaci *NetopeerGUI*, stanovil jsem přehled požadavků a cíle diplomové práce.

Kapitola 1 vysvětluje základní pojmy jako jsou protokol NETCONF a datové modely YANG. Uvádí přehled existujících řešení a shrnuje analýzu volby technologie provedené v bakalářské práci. Na základě konkrétních případů užití přináší návrh na rozšíření původní aplikace o novou funkcionalitu. Důležitými rozšířeními jsou např. modulárnost aplikace, vylepšené možnosti konfigurace nebo plný ajaxový průchod. V závěru je uvedeno shrnutí všech funkcí budoucí aplikace *v2.0*.

Kapitola 2 popisuje architekturu *NetopeerGUI*. Dále popisuje implementaci řešení případů užití zmíněných v analýze. Tato kapitola se zaměřuje na zajímavé a nestandardní části implementace, jako je práce s JavaScriptem, modulární rozšíření aplikace, vylepšení možností konfigurace a zpracování XML souborů v aplikaci. Nakonec popisuje zjednodušení procesu instalace a distribuce aplikace, včetně zveřejnění celého projektu na GitHub.

Kapitola 3 popisuje průběh a způsoby testování aplikace. Testování aplikace je rozděleno do 4 základních úrovní — jednotkové testy, akceptační testy, funkcionální testy a uživatelské testování. U každé úrovně testování jsou popsány okruhy testování a důvod, proč se daný typ testu hodí právě pro zvolenou část aplikace. Na závěr jsou uvedeny možné nápravy chyb objevené v uživatelském testování.

Kapitola 1

Analýza a návrh

1.1 Vysvětlení pojmů — NETCONF a YANG

1.1.1 Protokol NETCONF

Protokol NETCONF dle RFC6241 [4] „poskytuje prostředky k získávání stavových a konfiguračních informací, nastavení, manipulaci a mazání konfigurace síťových zařízení. Využívá k tomu kódování založené na *XML* (Extensible Markup Language), které je použito také pro zprávy protokolu. Operace protokolu jsou pak realizovány jako vzdálené volání procedur (*remote procedure calls - RPCs*).“

Podle [5] je NETCONF „navržen jako náhrada pro programová rozhraní založená na rozhraní příkazové řádky (Command Line Interface — *CLI*), jako je např. Perl[4] + Expect[5] přes Secure Shell (SSH). *CLI* bývá často používáno také lidmi, což zvyšuje složitost a snižuje předvídatelnost API pro reálné použití v aplikacích. Jako transportní vrstva je zpravidla používán protokol SSH. Konfigurační data zařízení jsou zakódovány pomocí XML — podrobněji v sekci 1.1.2.“

Všechna NETCONF zařízení musí podle [5] „umožnit zamknutí, editaci, uložení a odemknutí konfiguračních dat. Krom toho musí být všechny úpravy konfigurace uloženy během restartu v energeticky nezávislé paměti.“

Protokol (občas i samotná konfigurační data) je podle [5] „koncepčně rozdělen na základě „schopností“ (capability). Tyto schopnosti jsou identifikovány unikátním identifikátorem a předány serverem klientovi na začátku každé NETCONF relace.“ Obecně představují schopnosti seznam operací, které server podporuje.

Další zdroj [6] popisuje protokol NETCONF takto: „Protokol v sobě skrývá celou řadu otevřených standardů zaměřených na vzdálenou konfiguraci síťo-

[4] `http://www.perl.com`
[5] `http://expect.nist.gov`

vých prvků, serverů a služeb všeho druhu. Hlavní výhodou protokolu je nezávislost na výrobci zařízení a široká možnost automatizace.“

Protokol NETCONF je již nyní využíván např. oddělením nástrojů pro monitování a konfiguraci v organizaci CESNET. Je implementován u produktů komerčních firem jako jsou např. YumaWorks[6], Juniper[7] nebo MGSoft[8]. Příklad existujících implementací a nástrojů je uveden v sekci Přehled existujících řešení.

Seznam operací, které protokol NETCONF nabízí, je uveden včetně popisu v RFC 6241 [4] a RFC 6022 [7]:

- *<lock>* — uzamyká celé datové úložiště,
- *<unlock>* — odemyká datové úložiště,
- *<get>* — získává konfigurační a stavové informace,
- *<get-config>* — získává konfigurační informace,
- *<edit-config>* — upravuje data v konfiguraci úložiště,
- *<get-schema>* — získává schéma z NETCONF serveru.

Mod_netconf (více v sekci Architektura *NetopeerGUI*) obsahuje navíc interní příkazy pro komunikaci s NETCONF serverem.

- *<connect>* — připojuje se k *UNIX socketu*,
- *<disconnect>* — odpojuje se od *UNIX socketu*,
- *<info>* — získává informace o navázaném spojení.

1.1.2 Datové modely YANG/YIN

Datovým modelem se rozumí technický popis zařízení, který definuje jeho vlastnosti a vazby mezi jednotlivými komponentami zařízení, vytvořený výrobcem. Správce podle modelu dokáže zjistit, co všechno může editovat nebo jakou konfiguraci může vytvořit. Pro generování uživatelského rozhraní pak definuje důležité sémantické informace, které nejsou při komunikaci pomocí protokolu NETCONF potřeba a proto se u jednotlivých zpráv neuvádí.

K zápisu datových modelů se používá jazyk YANG (více informací v RFC 6020 [3]). Jedná se o logické členění zápisu v podobě stromové struktury. Existuje také alternativa, která používá syntaxi jazyka XML — formát YIN. Formát YIN nám umožňuje jednodušší strojové zpracování, protože nástroje pro práci s XML jsou dostupné téměř v každém programovacím jazyce.

[6] `http://www.yumaworks.com`
[7] `http://www.juniper.net`
[8] `http://www.mg-soft.si`

Interně se pro zpracování dat podle [5] využívá standardních XML nástrojů jako je např. XPath pro získání konkrétní podmnožiny konfiguračních dat. Všechny NETCONF zprávy jsou zakódovány v XML uvnitř XML jmenných prostorů (*XML Namespaces*).

1.2 Přehled existujících řešení

Jelikož protokol NETCONF i jazyk YANG jsou poměrně mladými technologiemi (první zmínka je v RFC4741 [1] z prosince 2006), není na trhu nástrojů pracujících s těmito technologiemi mnoho. Výčet známých implementací protokolu NETCONF je uveden v seznamu [8]. Pro srovnání s mou prací uvedu pouze implementace NETCONF klientů, kterým je také *NetopeerGUI* a jedná se tak o přímou konkurenci.

Seznam klientů obsahující CLI

- YUMAPro[9]
- NuDesign[10]
- Netopeer[11] — CLI, ze kterého volně vychází má práce

Seznam klientů nabízející GUI

- MG-SOFT NETCONF Browser Pro[12]
- NETCONFc[13]

Jako příklad zařízení uvedu produkty firem Juniper[14], CISCO[15] či CESNET — COMBO karta[16]. Protokol je používán např. pro konfiguraci systému Nemea[17]. Pro tento nástroj je v plánu připravit na míru vlastní modul, který bude využívat nově implementované modulární rozšíření *NetopeerGUI*.

1.2.1 YumaPro

Dle oficiálního popisu [9] je YumaPro[18] množinou nástrojů pro automatizaci vývoje a nastavení distribuovaných rozhraní jednotlivých zařízení. Nabízí CLI pro správu a testování NETCONF serveru [10].

[9] `http://www.yumaworks.com`
[10] `http://www.ndt-inc.com/`
[11] `https://www.liberouter.org/technologies/netconf/`
[12] `http://www.mg-soft.com/mgNetConfBrowser.html`
[13] `http://www.seguesoft.com`
[14] `http://www.juniper.net`
[15] `http://www.cisco.com`
[16] `https://www.liberouter.org/technologies/card-test/`
[17] `https://www.liberouter.org/nemea/`
[18] `http://www.yumaworks.com/products/yumapro/`

1.2.2 Netopeer

Netopeer je podle [11] projekt zaměřený na vzdálenou konfiguraci za použití NETCONF protokolu. Jedná se o projekt vyvíjený organizací CESNET. Obsahuje dílčí části *Netopeer server* a *Netopeer klient* postavené nad knihovnou *libnetconf*. *Libnetconf* je určena pro snadný vývoj NETCONF klientů a serverů.

NetopeerGUI využívá také knihovnu *libnetconf*. Více informací je uvedeno v následujících sekcích.

1.2.3 NETCONF Browser Pro

Oficiální popis aplikace [12] uvádí, že se jedná o uživatelsky přívětivého NETCONF klienta, který umožňuje získat, upravovat, instalovat a mazat konfiguraci libovolného NETCONF zařízení na síti.

Poskytuje intuitivní GUI, které umožňuje načíst jakýkoliv validní YANG či YIN modul a zobrazit ho v podobě hierarchického stromu. Umožňuje použití příkazů jako je *<get>*, *<get-config>*, *<edit-config>* a volání RPC metod. Dle dalších uvedených informací se jedná o univerzální aplikaci napsanou v jazyce JAVA. V době psaní tohoto textu se svou funkcionalitou stává jedinou přímou konkurenci *NetopeerGUI*.

Srovnání s *NetopeerGUI*

Celkový výčet funkčnosti uvedený na stránkách MG-SOFT NETCONF Browser Pro Features[19] se shoduje v seznamu podporované funkčnosti s *NetopeerGUI*. Jako příklad zmíním podporu datových modelů, všech operací protokolu NETCONF (*<get>*, *<get-config>*, *<get-schema>*, *<edit-config>*, volání RPC metod, validace...) a zobrazení NETCONF notifikací.

NETCONF Browser PRO ovšem také nabízí určité výhody, které jsou zatím v plánu na budoucí rozšíření *NetopeerGUI* (více v sekci Budoucí práce) — kompletní historie uživatelských příkazů v rámci jedné relace, načítání vlastních YANG modelů a podmíněné odeslání změn konfigurace.

Samotné uživatelské rozhraní *NetopeerGUI* nabízí stejné možnosti, ale v jiné formě. Rozdíl je hlavně ve způsobu zobrazení dat uživateli. NETCONF Browser Pro připomíná textový XML editor, ve kterém je nutné všechny příkazy „psát ručně“ (editor automaticky napovídá možné názvy a hodnoty). Oproti tomu *NetopeerGUI* transformuje a upravuje data z XML do uživatelsky přívětivé podoby, nezobrazuje tedy přímo daný XML soubor. Odstiňuje také uživatele od samotných příkazů protokolu NETCONF, které volá interně — ztrácí se nutnost znát rozdíly mezi *<get>* a *<get-config>* (a ve kterých situacích volat ten či onen příkaz), znalost syntaxe pro filtrování obsahu, změnu datového úložiště apod.

[19] `http://www.mg-soft.com/mgNetConfBrowser-features.html`

Navíc obsahuje tato usnadnění:

- automatické spojení *<get>* a *<get-config>* odpovědi do jednoho stromu pro větší přehlednost,
- předvyplnění výchozí hodnoty,
- sémantické informace — nápověda a popis k jednotlivým prvkům, definice výčtu dostupných hodnot apod. (na základě dostupných datových modelů),
- automatické generování podstromu povinných hodnot,
- filtr jednotlivých modulů a sekcí,
- transformace a seskupení hodnot do formulářových prvků (např. výběrové boxy pro výběr z výčtu povolených hodnot),
- intuitivní vkládání, editace a mazání jednotlivých prvků.

Další nespornou výhodou je také fakt, že *NetopeerGUI* je vyvíjeno jako veřejně dostupný open–source projekt (více v sekci Jednoduchá instalace a distribuce) zdarma. Licence MG-SOFTu s garantovanými aktualizacemi na jeden rok stojí 990 € (částka uvedena k 13. 2. 2015). Musím také zdůraznit rozdíl mezi cílovou platformou — NETCONF Browser Pro je distribuován jako desktopová aplikace napsaná v jazyce JAVA, zatímco aplikace *NetopeerGUI* je nabízena jako univerzální webová aplikace, která může být spuštěna téměř na jakémkoliv serveru a může pracovat i jako služba pro vzdálenou správu libovolného zařízení. Ukázka propojení *NetopeerGUI* s několika zařízeními (NETCONF servery) je uvedena na obrázku 1.1.

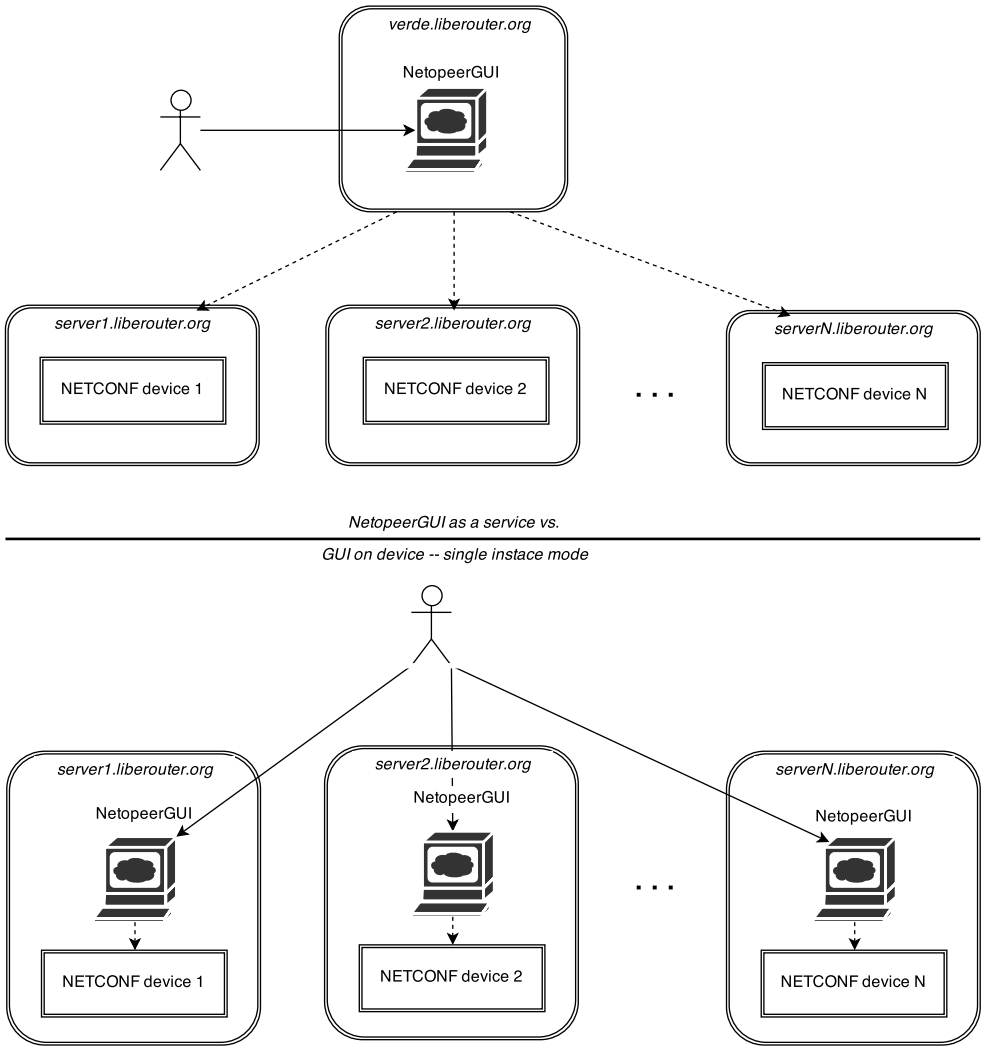
1.3 Volba vhodné technologie pro implementaci

Podrobný výběr vhodné technologie byl proveden již v rámci bakalářské práce [2]. Původní výběr technologií se při dalším vývoji osvědčil jako vhodný. V diplomové práci nedošlo k žádné změně a vývoj pokračuje za použití stejných technologií. Byly zvoleny standardní prostředky pro tvorbu webových stránek — programovací jazyk PHP pro aplikační vrstvu, značkovací jazyk HTML, JavaScript a CSS pro prezenční vrstvu.

Jako základ pro aplikační vrstvu byl zvolen framework Symfony2[20] společně s databázovým frameworkem Doctrine[21] pro ORM (objektově-relační modelování). Pro perzistentní vrstvu byla zvolena open–source databáze SQLite kvůli tomu, že nevyžaduje instalaci a tím pádem se nejedná o další závislost systému (perzistentní vrstva může být ale kdykoliv vyměněna uživatelem díky použití Doctrine).

[20] `http://www.symfony.com`

[21] `http://www.doctrine-project.org`



Obrázek 1.1: Způsoby připojení *NetopeerGUI* k NETCONF zařízení. Aplikace může fungovat ve dvou režimech — buď jako centrální služba pro konfiguraci N vzdálených zařízení (horní část obrázku), nebo v režimu jednoho zařízení, kdy je uživatelské rozhraní součástí zařízení.

Pro prezenční vrstvu je použit šablonovací systém Twig[22], který je výchozím šablonovacím systémem frameworku Symfony2. Nedílnou součástí prezenční vrstvy je také JavaScript a kaskádové styly CSS.

Hlavním úkolem pro aplikaci je generování GUI pro konfiguraci zařízení na základě datových modelů ve formátu YIN (informace kódovány ve standardním XML formátu). Cílem je transformovat odpověď NETCONF serveru

[22] `http://twig.sensiolabs.org`

obohacenou o informace obsažené v datovém modelu do podoby zobrazitelné ve webovém prohlížeči. Aplikační vrstva musí umět XML soubory zpracovávat a manipulovat s nimi. Většina programovacích jazyků, stejně jako PHP, obsahuje nástroje pro práci s XML. PHP nabízí dva základní objekty, pomocí kterých lze XML načíst a dále zpracovávat — SimpleXML a DOM. Při zpracování XML používám volně oba tyto objekty dle potřeby, pro výpis XML stromu v šabloně využívám vždy SimpleXML.

JavaScript

V *NetopeerGUI* bude potřeba použít pro snížení komunikace a datové náročnosti skriptovací jazyk, který bude spuštěn na straně klienta, tedy v prohlížeči uživatele. Jako příklad uvedu možnost editace vypsaného XML stromu či asynchronní požadavky pro aktualizaci obsahu bez nutnosti znovu načtení celé stránky. Skriptování je podle [13] „program, který nepotřebuje kompilaci před jeho spuštěním. V kontextu webového prohlížeče se jedná většinou o program napsaný v JavaScriptu (dále JS), který je vykonán prohlížečem, jakmile je stránka stažena nebo v průběhu práce uživatele se stránkou jako odpověď na událost vyvolanou uživatelem. Skriptování umožňuje vytvářet dynamičtější stránky, například bez nutnosti obnovení stránky může změnit její obsah.“ AJAXem rozumíme podle shrnutí uvedeném v článku [14] „asynchronní načítání XML dokumentů pomocí JavaScriptu. Nejedná se o novou technologii, ale pouze o spojení stávajících technologií do jedné nové (využití standardního HTML a CSS, DOMu (vysvětleno níže), XML a XSLT, XMLHttpRequest a JavaScriptu).“

Základním skriptovacím rozhraním vyvinutým v organizaci W3C[23] je DOM (Document Object Model). DOM podle [13] umožňuje programům a skriptům dynamicky přistupovat a aktualizovat obsah, strukturu a styly dokumentu. Tento popis naprosto vystihuje požadavky, které si na skriptování na straně klienta klademe.

V dnešní době stále bohužel neinterpretují všechny prohlížeče JavaScript stejným způsobem (i když se situace stále lepší). Pro odstínění od chování jednotlivých prohlížečů vznikly různé JS knihovny, jako je např. jQuery[24], MooTools[25] a mnoho dalších. *NetopeerGUI* nyní využívá knihovnu jQuery, která byla zvolena v původním výběru technologií.

Kaskádové styly — SASS

Kaskádové styly (CSS) jsou podle [15] jednoduchý mechanismus pro přidávání stylů (např. písma, barev, pozadí) webovým dokumentům. Jedná se o oddě-

[23] `http://www.w3.org`
[24] `http://www.jQuery.com`
[25] `http://www.mootools.net`

lení vzhledu stránky od samotného HTML dokumentu. CSS prošlo dlouhým vývojem a ustálilo se dnes ve verzi CSS3.

Po nástupu CSS3 došlo k podobné situaci, která nastala i u JavaScriptu — nekompatibilita zápisu mezi jednotlivými prohlížeči. Jednotlivé prohlížeče řeší zápis nových CSS3 vlastností většinou přidáním vlastních prefixů (např. -webkit-, -o-, -ms-, -moz-). Složitější zápis pomohl většímu rozšíření CSS preprocesorů.

Preprocesory obsahují základní programovací konstrukty jako jsou proměnné, cykly a funkce. Tím se stává přístup k zápisu CSS zcela odlišný, než dříve. Usnadnění zápisu je také umocněno tím, že se upravila syntaxe. Jako nejdůležitější považuji možnost zanořování jednotlivých elementů do sebe (*nesting*). Mezi základními funkcemi oceňuji zejména práci s barvami (např. ztmavení) či matematické operace. Tyto preprocesory nám obecně umožňují odstínit se od složitých zápisů prefixů a zbytečného duplikování kódu, podobně jako frameworky. Nutné je ovšem vytvořit si tyto funkce pro nové CSS3 vlastnosti — opět i zde ale existují frameworky, které tyto funkce řeší za nás.

V původní analýze byl zvolen preprocesor SASS[26] za použití frameworku Compass[27].

1.4 Návrh na rozšíření původní aplikace (*v1.0*)

V následujících sekcích se pokusím shrnout nedostatky aplikace *v1.0* na konkrétních případech užití (*use cases*). U každého případu užití tak definuji problém, který řeší aplikace *v1.0* buď nedostatečně, nebo vůbec. Následně navrhnu řešení problémů definovaných v případech užití v podobě rozšíření a vylepšení určené pro verzi *v2.0*.

1.4.1 Režim jednoho zařízení

Aplikace *v1.0* si kladla za cíl možnost připojení se k více zařízením najednou. Tento případ užití však není v praxi vždy potřeba — běžně může nastat situace, že se má *NetopeerGUI* připojit pouze k jednomu, předem definovanému, zařízení. Toto zařízení může být jak lokální, tak vzdálené. Jako příklad uvedu konfiguraci lokálního linuxového systému, nebo vzdálenou konfiguraci jednoho konkrétního nástroje (např. systému Nemea).

Řešením tohoto problému je rozšíření konfigurace aplikace o možnost nadefinování IP adresy a portu cílového zařízení, ke kterému se bude *NetopeerGUI* vždy připojovat. Tím pádem je také zbytečné mít dva přihlašovací formuláře (první do aplikace samotné, druhý pro připojení se k zařízení), stejně tak celou stránku s přehledem připojených zařízení. Proto by bylo vhodné, aby první formulář sloužil přímo pro připojení se k zařízení pomocí uživatelského

[26] `http://sass-lang.com`

[27] `http://compass-style.org`

jména a hesla — IP adresu a port už přeci známe z konfigurace. Pokud budou přihlašovací údaje správné, zobrazí se uživateli přímo detail zařízení.

1.4.2 Asynchronní načítání a notifikace

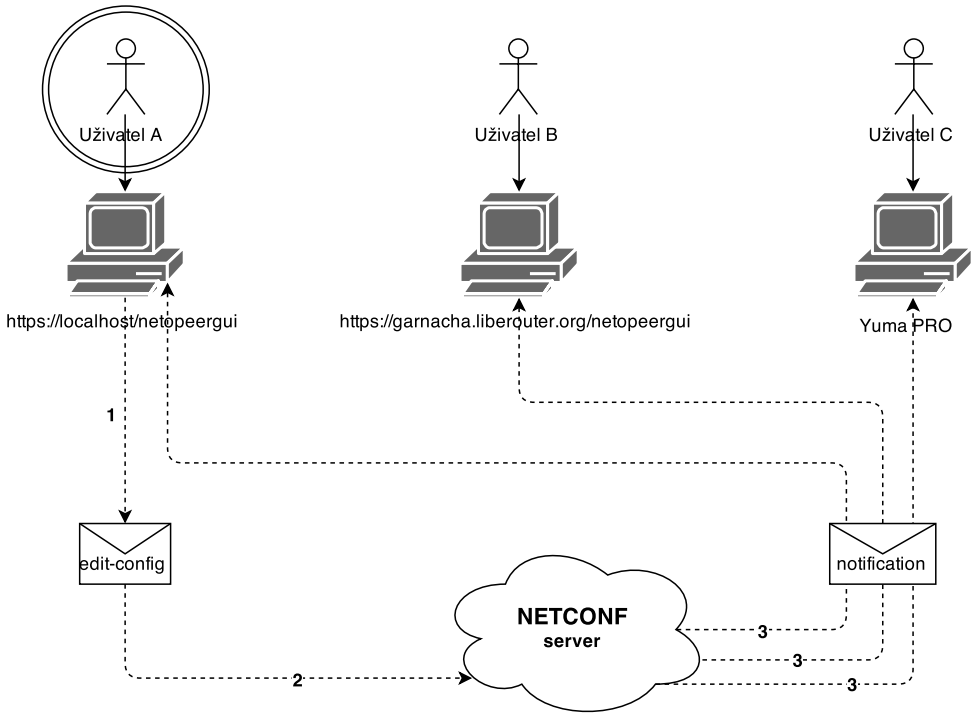
Novým požadavkem pro rozšíření aplikace *v2.0* jsou notifikace přenášené z NETCONF serveru, které nebyly ve verzi *v1.0* implementovány. Notifikacemi rozumíme informace o zařízení, ke kterému jsme aktuálně připojeni, typu:

- uživatel XXX z IP adresy 82.12.22.88 se připojil k zařízení
- uživatel XXX změnil konfiguraci zařízení
- vaše relace byla ukončena
- volání RPC metody bylo úspěšné
- odpověď na volání RPC metody: např. „*toast is done*“
- a další podobné informace

Notifikace protokolu NETCONF fungují následujícím způsobem. Představme si, že ve stejný čas konfigurují zařízení tři na sobě nezávislí uživatelé. Schéma této komunikace je uvedeno na obrázku 1.2. První uživatel provede v libovolný okamžik např. nějakou akci *<edit-config>*, která vyvolá vznik notifikace. Tato notifikace je doručena všem připojeným uživatelům, kteří tyto notifikace odebírají. Notifikace odesílá NETCONF server, nikoliv aplikace, ta je pouze příjemcem těchto notifikací. *NetopeerGUI* se stará o to, aby vypsalo všechny notifikace, které ze zařízení příjdou a zobrazilo je cílovým uživatelům. Je potřeba najít způsob, kterým bychom měli o této události uživatele informovat.

Hlavní problém je, jakým způsobem předat notifikace ze serveru do prohlížeče. Každý uživatel by se mohl v určitých intervalech (třeba jednou za 2 s) dotazovat serveru, jestli neobdržel novou nofitikaci. Problém je v tom, že server bude neustále dotazován ze strany klienta, jestli pro něj neexistuje nová notifikace. Bude tak vznikat velké množství HTTP požadavků. Notifikace je navíc nutné zobrazit uživateli ideálně ihned poté, co jsou vytvořeny. Interval 2 s může být zbytečně dlouhým.

Druhou alternativou je použití HTML5 Websockets (dále WS). Zdroj [16] popisuje WS takto: „WS jsou pokročilou technologií, která umožňuje otevřít interaktivní komunikační relaci mezi prohlížečem uživatele a serverem. S tímto API můžete posílat zprávy serveru a přijímat událostmi řízené odpovědi bez nutnosti opakovaného dotazování se serveru pro získání odpovědi v reálném čase“. Obecnými přednostmi WS jsou nižší režie, asynchronní komunikace, persistentní spojení, redukce objemu dat a možnost využití komunikačního kanálu pro interaktivní komunikaci.



Obrázek 1.2: Schéma vzniku notifikace. Ke stejnému zařízení jsou připojeni tři na sobě nezávislí uživatelé. Uživatel **A** vytvoří požadavek *<edit-config>*. NETCONF server tento požadavek zpracuje a o výsledku informuje všechny připojené uživatele **A**, **B** a **C** pomocí notifikace.

Klient se v prohlížeči připojí k socketu a v případě události, kdy dostane novou zprávu, ji zpracuje a vypíše uživateli. O předání notifikací od NETCONF serveru klientům se stará *mod_netconf*. S použitím WS ale přichází jedna komplikace. Klient se bude připojovat k socketu pro odběr notifikací při každé změně URL, tedy při téměř každém HTTP požadavku. Jelikož budou notifikace po přijetí pouze vypsány pomocí JS, přijde také o historii doručených notifikací. Tento problém se dá vyřešit pomocí plného ajaxového průchodu aplikací.

Ajaxový průchod aplikací

Aplikace *v1.0* nabízí klasický průchod aplikací — kliknutím na odkaz nebo odesláním formuláře přejde uživatel na jinou stránku aplikace, která se celá vykreslí. Představme si ale situaci, kdy při přechodu mezi jednotlivými stránkami aplikace dochází jen ke změnám určitých částí obsahu, ostatní části jako je např. menu zůstávají stále stejné. Našim cílem je, abychom při každém HTTP požadavku mohli vykreslovat pouze ty části stránky, které se změnily.

Není totiž nutné pokaždé načítat všechny JS, CSS nebo obrázky, vykreslovat celý DOM nebo registrovat JS události. Nemluvě o zpracování logiky pro načtení dat potřebných pro každou část obsahu. Problém nastane v případě, kdy potřebujeme uchovávat kromě stejných obsahových částí také stav aplikace a hodnoty globálních proměnných prostředí nebo zachovat spojení pro získání notifikací pomocí WS.

Řešením tohoto problému je ajaxový průchod aplikací, kdy nedochází k přechodu mezi stránkami kliknutím na odkaz, následné změně URL v prohlížeči a načtení nové stránky, ale pouze načtení změn určitých částí stránky (dále budu označovat jako invalidaci bloků). Tento přístup nabídne uživateli subjektivně rychlejší odezvu aplikace, protože nedochází k „problikávání“ při přechodu mezi stránkami a serveru ušetří výpočetní výkon. Nicméně pro přechod mezi stránkami aplikace je vhodné v HTML stále používat odkazy a formulářové akce, které definují cílovou URL stránky. URL funguje jako unikátní identifikátor zdroje na webu od svého počátku. URL adresa je jako identifikátor zdroje čitelná i pro uživatele, kteří si ji mohou např. uložit do záložek nebo poslat emailem.

Před příchodem HTML5 bylo možné měnit pomocí JS URL stránky v části za znakem #, tedy vlastnosti *window.location.hash*. Pro případný ajaxový průchod se používala změna hodnoty za hashem, což má své výhody. Změnu za hashem podporují všechny prohlížeče a fungují tlačítka zpět a vpřed. Problém ale nastává v routování jednotlivých požadavků v PHP. Část URL za hashem se totiž serveru nepředává. Když si uživatel zobrazí stránku s hashem, aplikace není bez použití JS schopna zjistit, kterou stránku má vykreslit. Představme si, že chceme pro generování odkazů na stránce použít výchozí metody frameworku Symfony2 pro generování URL odkazu — pojmenovanou cestu (dále v textu bude označováno jako *route*) a její parametry. Tímto způsobem je vygenerována URL, která nemá žádný hash, navíc nemá často ani stejný cílový soubor uvedený v REQUEST_URI (např. `/ajax/#/route/page.php` vs. `/route/page.php`). Tyto URL jsou pro zpracování v PHP zcela rozdílné. Při ajaxovém volání by proto musela existovat transformace formátů URL mezi sebou. Pokud se rozhodneme z nějakého důvodu JS vypnout, převod URL by mohl být komplikací.

V HTML5 naštěstí přibylo nové *HTML5 history API*. Zdroj [17] jej definuje takto. „*HTML5 history API* je standardizovaná cesta, jak manipulovat s historií prohlížeče pomocí JS. Část API — navigace historií — byla dostupná i v předchozí verzi HTML. Nová část uvedená v HTML5 obsahuje možnosti, jak přidávat záznamy do historie prohlížeče a zároveň viditelně měnit URL stránky také v liště prohlížeče (bez nutnosti znovunačtení stránky). Umožňuje také zpracovat záznamy, které byly vybrány z historie poté, co uživatel kliknul na tlačítko zpět v prohlížeči. Tím pádem URL může dělat přesně to, k čemu je určena — unikátně identifikovat zdroj i v aplikacích, které používají JS a neprovádějí plné načítání stránek.“

Toto API nabízí stejné možnosti, jako *window.location.hash* s tou výho-

dou, že je možné používat stejné URL adresy pro generování odkazů i jejich následné ajaxové načítání.

1.4.3 Rozšíření možností konfigurace

V aplikaci *v1.0* je možné konfigurovat zařízení dvěma způsoby — editací hodnot získaných operací *<get-config>* a nebo duplikací části podstromu (a editací duplikovaných hodnot). Představme si ale situaci, že máme hodnotu získanou *<get-config>* prázdnou nebo chybí potřebná část podstromu, kterou bychom chtěli duplikovat. Možnost pouze editovat hodnoty nebo duplikovat části podstromu je značně limitující.

Řešením je rozšíření možností konfigurace v aplikaci *v2.0* o kompletní množinu CRUD operací, v našem případě přidání možnosti vytvářet libovolné podstromy a také mazat vybrané uzly stromu.

Jednodušším úkonem je mazání uzlu (včetně jeho podstromu). Pro operaci *<edit-config>* je nutné přidat k uzlu, který chceme smazat, atribut *xc:operation = 'remove'*.

Složitější situace nastává u vytváření podstromů. Každému uzlu, který má v datovém modelu uvedeno, že se jedná o konfigurační uzel (tedy ten, který obdržíme z operace *<get-config>*), chceme umožnit vložení validního podstromu. Prvním krokem bude přidání tlačítka pro přidání potomka. Po kliknutí na tlačítko se automaticky vygenerují dva *<input/>* elementy — jeden pro název, druhý pro hodnotu. Takto je možné rekurzivně pokračovat do dalších úrovní nebo je možné přidávat uzly na stejné úrovni (kliknutím na tlačítko přidat potomka u rodiče).

Zobrazení struktury datového modelu

Předchozí případ užití nám umožní vytvoření libovolných elementů v několika úrovních. Uživatel ale nyní nemůže vědět, jaké názvy elementů, případně jejich hodnoty, má vyplnit. Nepozná také, které uzly jsou povinné pro vytvoření validní operace *<edit-config>*. Nyní si musí tyto informace buď pamatovat, nebo mít zobrazený datový model např. v externím textovém editoru.

Bylo by proto vhodné přidat schéma struktury datového modelu, kterou by uživatel mohl mít stále otevřenou a na základě které by názvy a hodnoty mohl vyplňovat. Schéma struktury datového modelu může mj. ukázat, které položky jsou povinné a jakého jsou datového typu.

Automatická nápověda dostupných názvů

I když nyní uživatel vidí schéma datového modelu, stále musí názvy elementů vyplňovat ručně. Musí se také zorientovat ve schématu a určit, na které úrovni a u kterého elementu se zrovna v editaci nachází. Jelikož uživatel vidí schéma datového modelu, aplikace si jej může načíst také.

Uživateli usnadníme problém vypisování názvů elementů tím, že mu nabídneme automatické našeptávání možných hodnot názvů elementů na základě datového modelu. Automaticky identifikujeme pozici právě editovaného *<input/>* elementu a z datového modelu si načteme názvy jeho potomků. Tyto názvy pak prezentujeme uživateli v podobě našeptávače hodnot. Našeptávač by měl umět vyhledávat v seznamu dostupných hodnot a automaticky filtrovat neodpovídající výsledky. Uživateli by mělo stačit napsat pouze pár znaků z názvu elementu a potvrdit výběr kliknutím nebo stisknutím tlačítka ENTER.

Automatická transformace pole pro vložení hodnoty elementu

V minulém případě bylo zmíněno, že známe možné hodnoty názvů potomků na základě datového modelu. Kdyby se element s tímto názvem vypisoval v šabloně (mimo editaci) jako výpis odpovědi z operace *<get-config>*, umíme k tomuto elementu vypsat další sémantické informace — textovou nápovědu, příznak, jestli je uzel povinný, nebo jej také umíme transformovat na formulářový prvek pro editaci hodnoty. Formulářovému prvku umíme automaticky doplnit výchozí hodnoty, množinu dostupných hodnot (*<select/>*) nebo doplnit korektní datový typ. Z předchozího případu užití jsme pomocí našeptávače získali název uzlu, který bychom mohli použít pro vykreslení všech dostupných sémantických informací (stejně jako v šabloně).

Proto po změně názvu uzlu automaticky načteme všechny informace z datového modelu, včetně nápovědy a dalších sémantických informací. Jelikož máme logiku pro výpis každého uzlu vyřešenou již v šabloně, není vhodné tuto logiku opakovat zbytečně také v JS. Pomocí ajaxového volání si proto načteme celý vykreslený řádek uzlu z HTML šablony a vložíme jej do editačního okna.

Automatické doplňování povinných položek

V předchozích odstavcích bylo zmíněno, že některé uzly jsou pro vytvoření validní operace *<edit-config>* povinné. Nyní budeme umět na základě datového modelu napovídat názvy potomků, zobrazovat formulářové prvky obohacené o sémantické informace, ale každý uzel podstromu stále musíme přidávat po jednom. Některé uzly přitom musí být obsaženy ve validním *<edit-config>* požadavku vždy. Jedná se typicky o povinné položky pro určitou část podstromu a také o klíčové položky u elementů typu *list* a další.

Po výběru názvu nového uzlu proto vytvoříme strukturu jeho podstromu obsahující všechny povinné uzly automaticky. Rovnou je zobrazíme uživateli, který těmto položkám určí pouze potřebné hodnoty.

Zpracování více editovaných podstromů do jedné zprávy

V aplikaci *v1.0* se formulář pro duplikování podstromu vždy odesílal při každé jednotlivé duplikaci nebo po změně hodnoty uzlu stromu. Stejným způsobem bylo původně implementováno také vkládání nových podstromů. Přidávání podstromů tak bylo vždy odděleno od editace hodnot obdržených z *<get-config>*. Formulář s úpravami podstromu se po každé jednotlivé úpravě podstromu odesílal. V průběhu používání aplikace se však objevil případ užití, ve kterém bylo nutné přidat/editovat více podstromů najednou — konkrétně hovoříme o uzlech typu *leaf-ref*. Tyto uzly obsahují jako hodnotu referenci do některé jiné části konfiguračního stromu a může se stát, že je vyžadováno, aby byly oba uzly obsažené ve vygenerovaném požadavku *<edit-config>* současně. Tím pádem bylo nutné navrhnout operaci („commit“), která odešle celou modifikovanou konfiguraci namísto odesílání jednotlivých změněných částí postupně.

Jako řešení tohoto případu užití bylo navrženo, aby se formulář pro vytváření podstromů místo odeslání vložil do konfiguračního stromu mezi ostatní uzly. Všechny formuláře, které jsou takto vloženy, mají jiný atribut *name*, aby bylo možné je odlišit při zpracování od úprav hodnot existující konfigurace.

Podpora více datových úložišť

Aplikace *v1.0* umožňovala práci pouze s jedním datovým úložištěm obsahujícím aktuálně běžící konfiguraci, zvaným *running*. Ostatní datová úložiště nebylo možné nijak konfigurovat ani zobrazit. Pokud zařízení podporuje více datových úložišť (*datastore*), budeme muset nabídnout způsob, jak je spravovat a volně mezi nimi přepínat. Pro usnadnění práce bude potřeba umožnit kopírování konfigurací mezi jednotlivými datovými úložišti. Informace o aktuálním aktivním datovém úložišti si budeme muset pamatovat, protože je nutné tuto informaci předávat také jako parametr do požadavku *<edit-config>*.

Přepínání mezi jednotlivými datovými úložišti by mohlo být vyřešeno přidáním jednoduchého formuláře obsahující přepínač dostupných datových úložišť. Pro uložení názvu aktivního datového úložiště je možné použít, podobně jako pro jiné další filtry, proměnnou *$_SESSION*. Kromě výběru aktivního datového úložiště přidáme do formuláře také výběr cílového datového úložiště, do kterého chceme aktivní konfiguraci překopírovat.

Vyplnění prázdných datových úložišť a práce s prázdnými moduly

Na začátku sekce jsem zmínil, že aplikace *v1.0* uměla pouze editovat hodnoty získané pomocí operace *<get-config>*, případně uměla duplikovat části podstromů. Neexistoval žádný způsob, jak začít konfigurovat prázdný modul nebo datové úložiště.

Speciálním případem vytváření podstromů je založení celé konfigurace jednoho modulu. Pokud v nastavení NETCONF serveru povolíme nový modul,

neznamená to, že se zobrazí automaticky v horním menu mezi aktivními moduly. Horní menu je generováno na základě odpovědi *<get>*, ze které si načte kořenové elementy (což jsou názvy modulů). Po povolení modulu v NETCONF serveru ale odpověď *<get>* tento kořenový element neobsahuje. Musíme jej proto ručně vytvořit. Stejný problém se může objevit při přepínání mezi datovými úložišti, kdy je např. datové úložiště *candidate* prázdné a neobsahuje ani kořenový element.

Pro vytváření nových kořenových elementů musí být do požadavku *<edit-config>* vložen kromě názvu modulu také jeho správný jmenný prostor (*namespace*). Mapování jmenného prostoru s názvy modulů máme k dispozici z *capabilities* hlaviček, které jsme získali po připojení k NETCONF serveru. Pro vytváření nových kořenových elementů byl proto navrhnut jednoduchý formulář, který bude obsahovat dva prvky — název modulu a jeho jmenný prostor. Každý prvek bude obsahovat našeptávač s dostupnými hodnotami. Po výběru názvu modulu (kořenového elementu) se automaticky vyplní příslušný jmenný prostor tak, aby uživatel nemusel tuto informaci nikde složitě hledat.

Uživatelem řazené položky

U uzlů datového typu *list* nebo *leaf-list* umožňuje YANG definovat, že záleží na pořadí jejich potomků. To umožňuje popsat pravidla *firewallu* nebo záznamů ve směrovacích tabulkách apod. Interně se uživatelsky řazené seznamy určují pořadím uzlů potomků v požadavku *<edit-config>*. Aplikace *v1.0* tuto funkcionalitu nenabízí. Pro aplikaci *v2.0* by bylo vhodné umožnit uživateli na frontendu určit pořadí potomků — jejich seřazení.

Základním řešením by mohlo být u každého potomka (pouze pro uzly s rodičem typu *list* a *leaf-list*) doplnění tlačítka se šipkami nahoru a dolů. Kliknutím na šipku by se potomek (včetně jeho podstromu) posunul o jednu pozici. Tento postup je relativně zdlouhavý pro více položek seznamu. Vhodnou alternativou by mohl být jQuery plugin *Sortable*[28], který umožňuje přetahovat jednotlivé prvky seznamu na jakoukoliv pozici. V konfiguraci pluginu je možné nadefinovat omezení tak, aby nešlo přetahovat prvky mimo jejich rodiče (mezi jiné seznamy). S přetahovanými prvky je možné přenášet i jejich případné podstromy. Plugin *Sortable* se jeví uživatelsky přívětivějším, proto jsem se rozhodl pro jeho implementaci na úkor jednoduššího posunu potomků o jednu pozici.

U obou dvou přístupů bude nutné předat informace o pořadí jednotlivým formulářovým prvkům tak, aby bylo možné jednoduše seřadit potomky seznamů při dalším zpracování v PHP.

[28] https://jqueryui.com/sortable/

Volání RPC metod

Každý modul může v datovém modelu definovat RPC metody, které nabízí. U příkladu *toasteru* může jít např. o metodu „make toast“ [18]. Tato metoda může mít předem definované vstupní parametry, jako je např. barva kůrky nebo odložený start a také akci „START“, neboli akci spusť proces (zavolej RPC metodu). V aplikaci *v1.0* se tato část datového modelu nijak nezpracovávala, tím pádem ani uživatel neměl žádnou možnost RPC metody volat.

Pro rozšíření v aplikaci *v2.0* zobrazíme uživateli seznam dostupných RPC metod podle datového modelu v levém panelu. Po kliknutí na název RPC metody se otevře modální okno s předvyplněnou strukturou povinných parametrů na základě datového modelu (nemusí být žádné). Parametry jsou zobrazeny ve formě formulářových prvků, včetně všech dostupných sémantických informací — jedná se pouze o jiný způsob výpisu XML stromu, který již známe. Modální okno bude možné zavřít kliknutím na křížek nebo potvrdit volání metody odesláním obsaženého formuláře.

1.4.4 Modulární rozšíření aplikace

Univerzální řešení jednoho problému nemusí být vždy ideální a stoprocentní pro všechny situace. Je možné, že konkrétní zařízení potřebuje nabídnout kromě konfiguračních dat také např. vizualizaci, graf, obrázek, nebo jen jinou než textovou formu výpisu konfigurace. Univerzální řešení nemůže pokrýt všechny tyto individuální požadavky. Na řadu tak přichází otázka, jak výrobcům umožnit promítnout do *NetopeerGUI* jejich nestandardní, individuální požadavky.

Individuální požadavky na formu výstupu konfigurace mají řešení v podobě modulárního rozšíření *NetopeerGUI* tak, aby měli výrobci plnou kontrolu nad formou výstupu svých modulů. Možnost modulárního rozšíření však přináší několik otázek a problémů.

Výběr způsobu výpisu

Bude nutné nějakým způsobem určit, kterou šablonu máme pro aktivní modul vykreslit (výchozí zůstane klasický výpis — tabulkový výpis). Nezbytná proto bude databázová struktura, ve které bude definováno mapování modulu na šablonu. Databázová struktura by měla mapovat název modulu (společně s jmenným prostorem) k *1..n* možným šablonám (formám výstupů).

Pokud bude modul nabízet vlastní šablonu pro výpis, měla by zůstat zachována i možnost přepnutí na výchozí šablonu. Pro volbu způsobu výpisu (tedy vzhledu modulu) stačí do levého panelu přidat přepínač se seznamem dostupných vzhledů. Po výběru jiného vzhledu se změna promítne do konfigurační části aplikace.

Způsob uložení individuálního vzhledu

Každá šablona by měla být umístěna v nějakém svém unikátním jmenném prostoru, aby byla možná její nezaměnitelná identifikace. Pro způsob uložení šablon se přímo nabízí využití vlastního Symfony2 Bundle (dále v textu budu označovat pouze jako *bundle*). *Bundle* bude obsahovat veškeré soubory (kontrolery, šablony, JS, CSS a obrázky) potřebné pro zobrazení individuálního vzhledu modulu. Autorovi rozšíření nabídne určitou volnost a možnost přetížení funkcí, tříd, metod nebo služeb a ovlivnit tak chování celého modulu. Šablony budou pod jmenným prostorem *bundle* také identifikovány v databázi.

Aby bylo možné dodržet tuto nově nadefinovanou architekturu, bude muset být výchozí vzhled definován také jako samostatný *bundle*. Aplikace si pro způsob výpisu konfigurace bude volit pouze konkrétní *bundle*, který se bude muset o celý výpis postarat sám.

Oddělení společných částí aplikace od individuálních

Obecně bude potřeba pro přechod k modulární architektuře aplikace rozdělit také JS, CSS a obrázky (dále *assets*) – není důvod načítat zdrojový kód a soubory, které jsou pro daný výpis zcela zbytečné. S tím souvisí nutnost rozdělení *assets* do jednotlivých funkčních bloků — *bundle*. Bude nutné oddělit obecný vzhled a layout aplikace od konfigurační části, kterou bude nově celou reprezentovat samostatný *bundle*.

Pro CSS bude nutné přepsat veškeré společné vlastnosti do konkrétních proměnných tak, aby je mohly nové *bundle* používat pro definici vlastního vzhledu (např. primární barvu tlačítek apod.). Definice společných vlastností v podobě proměnných umožní také „přizpůsobení“ celého vzhledu aplikace přetížením těchto výchozích proměnných a následnou překompilací stylů.

Pro všechny *assets* potřebné pro konkrétní *bundle* bude potřeba připravit jednotnou cestu, jak tyto soubory definovat a z prostředí celé aplikace je automaticky načítat. Zde se dá využít možností přetížení bloků u TWIG šablon. Nadefinujeme proto šablonu rozšiřující výchozí šablonu vzhledu stránky společně s definicí bloků (TWIG blocks), které bude mít možnost autor *bundle* přetížit a tím doplnit cesty k potřebným *assets*. Z prostředí *bundle* bude mít také možnost přetížit bloky celého layoutu, pokud to bude jeho cílem (např. skrytí bloku horního menu).

Příprava rodičovského kontroleru a API

Každý *bundle* musí mít možnost nějakým způsobem načítat konfigurační data pro vykreslení ve svých šablonách. V aplikaci *v1.0* se o přípravu dat a vykreslování všech informací o připojeném zařízení stará jeden kontroler – *DefaultController*. To není pro výpis z ostatních *bundle* optimální, protože tento kontroler je těžce rozšiřitelný. Chceme také ulehčit práci ostatním vývojářům

rozšiřujících *bundle* a zamezit nutnosti duplikace zdrojového kódu pro přípravu dat. Z tohoto důvodu bude muset dojít k rozdělení *DefaultControlleru* na *DefaultController* a *ModuleController*.

V *DefaultControlleru* zůstává všechna logika, která se stará o vykreslení společných částí pro všechny *bundle*. Rozšiřující *bundle* by neměly mít potřebu do tohoto kontroleru nijak zasahovat, či jej rozšiřovat. Rozšiřující *bundle* se stará pouze o výpis konfigurace modulu. K tomu bude sloužit obecná akce *moduleAction*, která bude nadefinována v novém *ModuleControllerInterface*. Toto rozhraní budou muset implementovat všechny rozšiřující *bundle*. Pro zjednodušení práce může nový kontroler v rozšiřujícím *bundle* přetížit nově vzniklý *ModuleController* a využívat tak jeho metody pro přípravu dat. Celá architektura je zobrazena na obrázku 1.3.

Z každého kontroleru (i z rozšiřujícího *bundle*) budou k dispozici služby pro komunikaci s *mod_netconf* či pro zpracování XML souborů (více podrobností v sekci Architektura *NetopeerGUI*). Tyto služby nabídnou API, které může využívat každý *bundle* pro vlastní způsob zpracování dat a komunikaci se zařízením, pokud bude chtít.

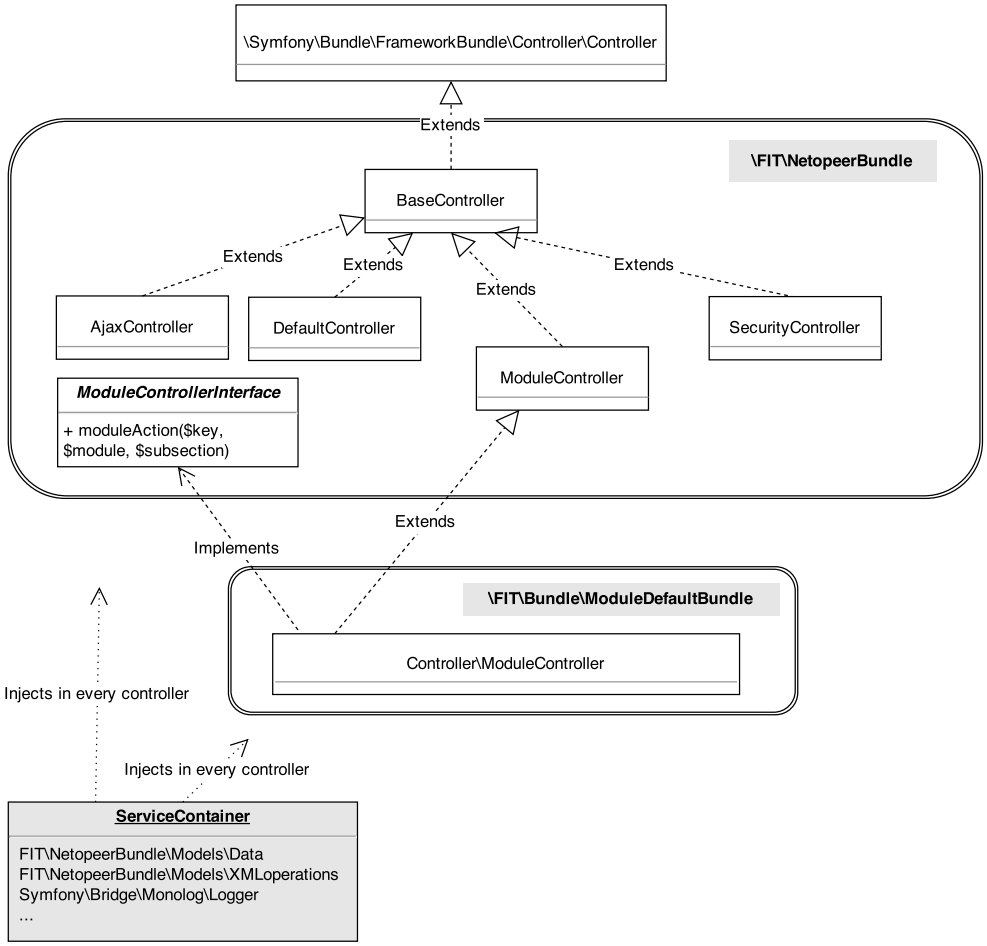
Výběr *bundle* pro zpracování akce

Akce *moduleAction* má definované pojmenované cesty (*route*), které jsou společné pro všechny rozšiřující *bundle*. Aplikace samotná musí nějakým způsobem ovlivnit načítání *bundle* pro cestu, která patří k této akci. Tím bude zajištěno, že se vždy vypíše aktivní modul pomocí konkrétního *bundle*. Pro mapování modulu a *bundle* máme navrženou databázovou strukturu. Zbývá tedy navrhnout, jak naprogramovat výběr namapovaného *bundle*.

Symfony nabízí možnost přetížit zpracování požadavku ještě před voláním výchozího kontroleru pomocí metody *onKernelController* ve vlastním *Listeneru*. V dokumentaci [19] je uvedeno, že metoda *onKernelController* obsahuje logiku, kterou chceme vykonat těsně před spuštěním samotného kontroleru. Toto je ideální místo, ve kterém se můžeme rozhodnout, který *bundle* (a konkrétní kontroler), budeme chtít pro výpis akce *moduleAction* zavolat. Pokud má požadavek přiřazenou akci *moduleAction*, načteme si z databáze podle názvu modulu a jmenného prostoru hodnotu možných kontrolerů pro výpis. Pokud nějaký záznam nalezneme, přepíšeme výchozí cílový kontroler na námi nově načtený z databáze. Celou akci necháme dále proběhnout, ale již v konkrétním *bundle* pro daný modul.

1.4.4.1 Další rozšíření

Další rozšíření aplikace se budou týkat hlavně nastavení možností aplikace. Toto nastavení bude možné provést buď v průběhu instalace, nebo také později nastavením parametrů v souboru *parameters.yml*. Uživatel si bude moct zvolit



Obrázek 1.3: Návrh architektury modulárního rozšíření pro vlastní způsob výpisu modulů.

např. jako výchozí režim aplikace režim jednoho zařízení nebo autentizaci pomocí SAML protokolu.

Jakékoliv modulární rozšíření, které bude distribuováno v podobě Symfony2 Bundle, bude možné přidat jako závislost do *composer.json* (více informací v sekci 2.9) a následně jej nainstalovat do aplikace. To nám umožní jednoduše celé *NetopeerGUI* rozšiřovat dle individuálních potřeb.

1.5 Celkový výčet funkcí aplikace (*v2.0*)

NetopeerGUI obsahuje následující funkce:

- připojení se k jednomu zařízení podle údajů uvedených v konfiguraci

aplikace — režim jednoho zařízení (např. lokální konfigurace linuxového systému),

- nebo připojení se k více lokálním i vzdáleným zařízením najednou (pomocí protokolu NETCONF) a volně mezi nimi přepínat,
- zobrazení historie připojených zařízení,
- podpora více datových úložišť, včetně kopírování konfigurací mezi nimi,
- zamykání a odemykání zařízení (*<lock>* a *<unlock>* operace),
- zobrazení stránky s informacemi o zařízení (capabilities...),
- validace obsahu datového úložiště,
- záloha konfigurace zařízení,
- podpora *<get-schema>*,
- automatické rozdělení přijatých stavových a konfiguračních informací do modulů a záložek (filtrace odpovědi),
- zobrazení výstupu operací *<get>* a *<get-config>* po kliknutí na zvolený modul (bez nutnosti ručního použití protokolu NETCONF),
- automatické doplnění výstupu operací *<get>* a *<get-config>* o sémantické informace na základě datových modelů YANG,
- transformace výstupu do podoby formulářových prvků pro snadnou editaci (možnost editace hodnot jednotlivých elementů), včetně zobrazení vhodných formulářových prvků na základě datového typu,
- vytváření nových modulů a vkládání nových elementů, včetně vytvoření kompletního stromu elementů,
- automatické napovídání názvů elementů a automatické doplňování povinných položek při vytváření stromu elementů, automatické doplnění výchozích hodnot,
- uživatelské řazení elementů (těch, které mají tuto možnost povolenou v datovém modelu),
- automatické provedení operace *<edit-config>* po uložení úprav,
- validace uživatelského vstupu,
- volání definovaných RPC metod,
- zobrazení asynchronních notifikací NETCONF serveru,

- záloha zvolené konfigurace.

Z hlediska univerzálního použití a možnosti rozšíření v komunitě uživatelů NETCONF umožňuje aplikace použití následujících funkcí:

- jednoduchá instalace pomocí *composer*,
- distribuce stažitelného předinstalovaného virtuálního stroje,
- přihlášení uživatelů do aplikace podle údajů uložených v databázi (lokální uživatelé aplikace), nebo pomocí SAML standardu,
- modulární rozšíření — každý modul může mít svou vlastní formu výstupu, každý výrobce si může upravit vzhled celé aplikace,
- modulární rozšíření je realizováno pomocí Symfony2 Bundle,
- těmto *bundle* je zpřístupněna služba (API) pro volání operací protokolu NETCONF, služba pro zpracování XML souborů a přípravu *<edit-config>* požadavku (úprava stromu na základě modifikovaného obsahu),
- *bundle* mohou jednoduše rozšiřovat třídy i služby aplikace dle potřeby.

Kapitola 2

Realizace

2.1 Architektura aplikace

2.1.1 Propojení NETCONF klienta s NETCONF serverem

Architektura NETCONF klienta je vícevrstvá. Podle [20] komunikuje NETCONF klient s NETCONF serverem pomocí knihovny *libnetconf* napsané v jazyce C. Tato knihovna implementuje protokol NETCONF a poskytuje základní funkce pro vytvoření spojení, přijímání a odesílání zpráv a práci s konfiguračními daty. Spojení využívá protokol SSH. Názorně je komunikace zobrazena na obrázku 2.1.

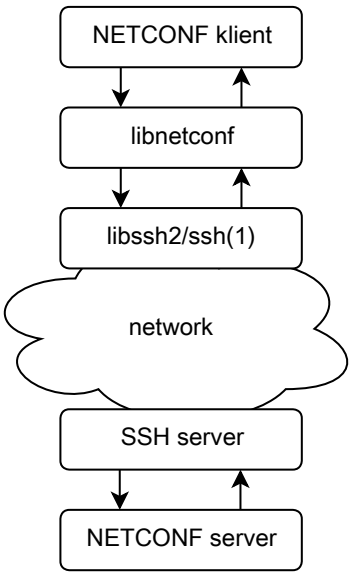
2.1.2 Architektura *NetopeerGUI*

NetopeerGUI rozšiřuje schéma uvedené na obrázku 2.1 o *mod_netconf*, jak je uvedeno na obrázku 2.2. *Mod_netconf* je modul pro server Apache[29]. S tímto modulem probíhá komunikace z webové aplikace přes *UNIX socket*. Modul využívá *libnetconf*, jak je uvedeno v hierarchii na obrázku 2.2.

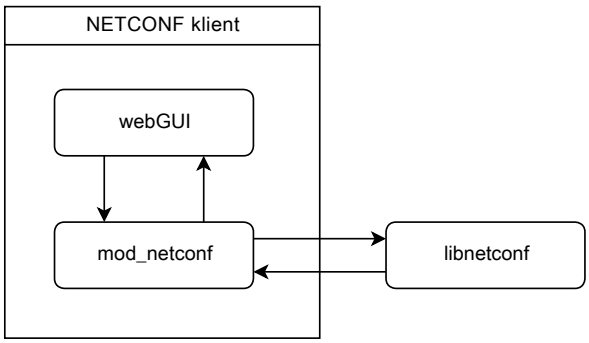
Veškerou obsluhu komunikace s *mod_netconf* provádí jedna třída *Data*. Komunikace třídy *Data* je znázorněna na obrázku 2.3. Tato třída zajišťuje provedení základních příkazů, které můžeme v rámci protokolu NETCONF použít. Výchozí metodou pro veškerou práci a komunikaci s *mod_netconf* je metoda *handle()*. Umožňuje provést operace, které API *mod_netconf* poskytuje. Odpovědi NETCONF příkazů ve formátu XML služba parsuje a dále zpracovává pro další použití v kontrolerech a šablonách.

Třída *Data* musí být v rámci *NetopeerGUI* dostupná ze všech kontrolerů, protože provádí veškerou komunikaci s *mod_netconf*. Z tohoto důvodu jsem tuto třídu vytvořil jako *službu* (*Service*) v rámci návrhového vzoru *Dependency Injection* (DI)). DI podle [21] [22] „slouží pro snížení závislostí mezi jednotli-

[29] `http://httpd.apache.org`



Obrázek 2.1: Diagram vrstev aplikací mezi NETCONF klientem a NETCONF serverem — konfigurovaným zařízením, které spolu komunikují pomocí knihovny *libnetconf* [20].

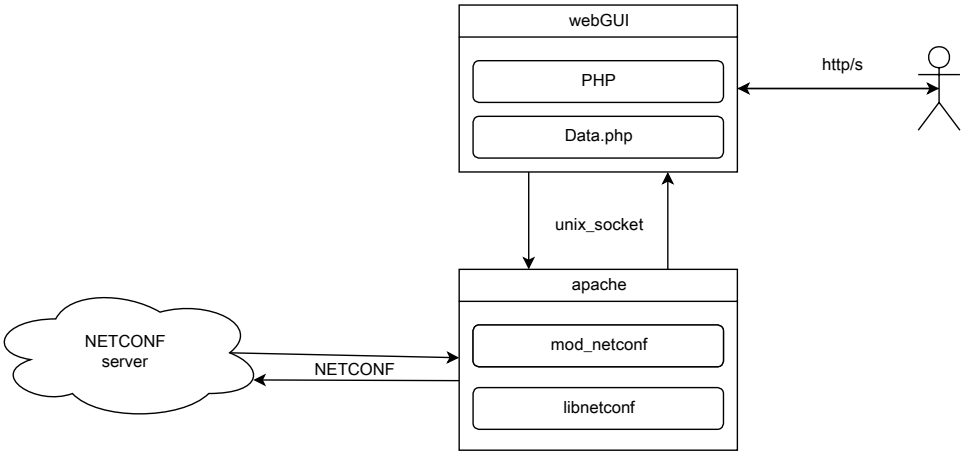


Obrázek 2.2: Zjednodušená architektura *NetopeerGUI* (NETCONF klienta) s *libnetconf* pomocí *mod_netconf*.

vými částmi systému. Jeho provedení vychází z obecnějšího návrhového vzoru *Inversion of Control* (IoC).“

Framework Symfony2 má pro definování služeb a jejich závislostí připravenou strukturu v souboru *Resources/config/services.xml*. Díky tomu je nyní možné používat třídu z jakéhokoliv kontroleru (či ji vložit do jiné třídy (nutné přidat závislost)) bez nutnosti vytváření konkrétních instancí přímo v kódu jednotlivých metod.

Definováním třídy *Data* jako služby jsem vytvořil základ API pro mo-

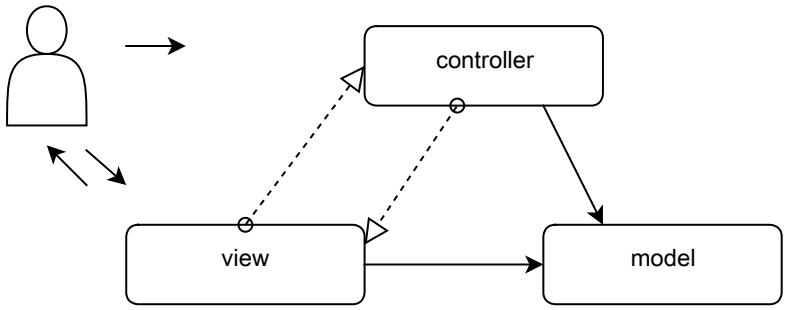


Obrázek 2.3: Podrobná architektura *NetopeerGUI* (NETCONF klienta) s NETCONF serverem.

dulární rozšíření (více v sekci Modulárnost aplikace). Dále v textu bude tato služba označována jako *DataModel* (se stejným názvem je nadefinovaná i v aplikaci).

Aplikování a práce s MVC

V článku „Úvod do architektury MVC“ [23] je návrhový vzor MVC popsán takto: „Architektura MVC dělí aplikaci na 3 logické části tak, aby je šlo upravovat samostatně a dopad změn byl na ostatní části co nejmenší. Tyto tři části jsou Model, View a Controller. Model reprezentuje data a business logiku aplikace, View zobrazuje uživatelské rozhraní a Controller má na starosti tok událostí v aplikaci a obecně aplikační logiku.“ Názorná komunikace je uvedena na obrázku 2.4.



Obrázek 2.4: Názorná ukázka komunikace mezi jednotlivými vrstvami v aplikace založené na návrhovém vzoru MVC.

Při používání a snaze o dodržování *best practices* pro vytváření aplikací v rámci Symfony2 ale přijdeme na to, že se striktně dle definice o třívrstvou MVC architekturu nejedná. Architekturu bych nazval podle zkušeností získaných v předmětu BI-ZSI spíše „dvou a půl vrstvou“, protože Controller je často nucen řešit také business logiku aplikace. Celý framework je silně závislý na používání *jmenných prostorů (namespaces)*, které poskytuje PHP od verze 5.3 [24].

2.2 Zpracování XML souborů

NetopeerGUI pracuje s XML soubory jako primárním formátem dat. Na práci s XML bychom se mohli dívat dvěma pohledy — načtení odpovědi ze serveru a příprava pro další zpracování (např. výpis v šabloně) nebo vytvoření XML řetězce se změnami konfigurace zařízení pro odeslání zpět na server pomocí operace *<edit-config>* (také volání RPC metod). Veškerá logika je umístěna ve třídě *XMLOperations*.

2.2.1 Zpracování odpovědi ze serveru

V prvním kroku se pokusíme načíst odpověď serveru v podobě XML řetězce pomocí *SimpleXML* objektu. Pokud načtení proběhne v pořádku, znamená to, že je XML řetězec je validní a lze jej dále zpracovávat. V opačném případě se často stává, že XML řetězec neobsahuje kořenový uzel. Pokusíme se proto obalit XML řetězec do chybějícího kořenového uzlu a zkusíme načíst XML znovu. Pokud ani tento pokus není úspěšný, oznámíme uživateli chybu při zpracování pomocí chybové hlášky a zpracování odpovědi zde končí — nemáme validní XML řetězec a nemůžeme jej ani vypsat.

Pokud je načtení odpovědi úspěšné a jedná se o operaci *<get>* či *<get-config>*, provedeme spojení odpovědi s předgenerovaným datovým modelem (získaným pomocí operace *<get-schema>* po navázání spojení se serverem) nebo vrátíme rovnou původní odpověď (pokud datový model neexistuje). Spojení odpovědi s datovým modelem doplňuje k odpovědi atributy se sémantickým významem pro výpis v šabloně. Příkladem takové sémantické informace jsou datové typy pro správnou transformaci do formulářových prvků, text nápovědy, informace, zda je hodnota povinná pro vyplnění apod.

V obou případech následují buď některé z dalších operací s XML odpovědí, nebo rekurzivní výpis do šablony.

Výpis XML do šablony

Výpis XML stromu do šablony probíhá pomocí rekurzivního volání několika šablon (zvlášť pro výpis rodičů a jejich potomků). Každý element je při výpisu zobrazen buď jako text nebo jako formulářový prvek (pokud má být

editovatelný) na základě informací uvedených v datovém modelu. U již existující konfigurace není zpravidla možné měnit název elementu, pouze jeho hodnotu (neplatí při vytváření nových částí stromu). Každý element je zobrazen v podobě jednoho řádku, který obsahuje navíc tlačítka pro vložení nového podstromu nebo jeho smazání.

Při výpisu se u jednotlivých elementů uchovává informace o XPath cestě, která je zakódována na základě přepisovacích pravidel (viz. dále). XPath cesta se používá pro jednoznačnou identifikaci elementu uvnitř XML stromu tak, aby bylo možné měnit konfigurační hodnoty, přidávat či odebírat elementy — libovolně pracovat s XML stromem. Hodnota XPath je přenášena jako název formulářového políčka (atribut *name*) a po odeslání formuláře je získána z pole *$_POST*. XPath cesta musí být proto zakódována do povolených znaků pro přenos v poli *$_POST*. XPath cestu je možné generovat automaticky pouze při prvním výpisu elementů do šablony, protože většina úprav XML stromu se následně provádí pomocí JS dynamicky. Při dynamickém vytváření již není k dispozici *SimpleXML* objekt a XPath cesta tak musí být generována přímo v JS.

2.2.2 Příprava XML řetězce pro operaci *<edit-config>*

Po odeslání formuláře s úpravami jednotlivých XML elementů je nutné odeslat serveru zprávu pomocí operace *<edit-config>* ve správném formátu. XML řetězec musí obsahovat požadované elementy a atributy a musí být ve striktně definované struktuře. Proto je nutné po úpravě hodnot jednotlivých elementů (či přidání podstromu apod.) provést další modifikace celého XML požadavku.

Jako příklad si můžeme uvést pročištění XML stromu pro smazání elementu. Nejprve z *$_POST* hodnot zjistíme, který element v XML stromu chceme smazat a vybereme jej pomocí XPath z aktuální konfigurace. K tomuto elementu doplníme atribut *xc:operation = 'remove'*. Operace *<edit-config>* musí být validní podle datového modelu, jinak se operace neprovede. To znamená, že musí obsahovat všechny požadované elementy a kompletní cestu od kořenového elementu. Ostatní elementy, kterých se této změny netýkají, odstraníme. Ponecháme pouze povinné elementy, které jsou zpravidla nutné pro unikátní identifikaci modifikovaného elementu.

Před odesláním zprávy pomocí operace *<edit-config>* je provedena také sémantická validace odesílaného XML řetězce pomocí Relax NG[30] a W3C XML Schema[31]. Soubory s validačními schématy jsou vygenerovány v průběhu generování datových modelů pro připojené zařízení.

Vkládání nových konfiguračních informací v datovém úložišti serveru se provádí podobným způsobem. I v těchto případech je potřeba vytvořit validní zprávu podle datového modelu.

[30] `http://relaxng.org`
[31] `http://www.w3.org/XML/Schema`

Operace „commit“

Operace „commit“ uvedená v analýze slouží k odeslání více úprav stromu v rámci jednoho požadavku. To s sebou přináší různé problémy. Při úpravách XML stromu pomocí JS vzniká několik menších formulářů, jejichž data jsou po odeslání formuláře serializována a dále zpracovávána v PHP. Každý formulářový prvek obsahuje v atributu *name* unikátní identifikátor formuláře. V hodnotě *$_POST* jsou proto jednotlivé formuláře také identifikovány unikátně.

Zpracování hodnot v PHP probíhá iterativní formou. Zpracování všech obdržených hodnot formuláře najednou by mohlo vést k mnoha kolizím při úpravách a modifikacích výsledného XML požadavku. Proto jsou data z každého jednotlivého formuláře zpracována postupně jeden po druhém, úplně stejně, jako kdyby byl odeslán pouze jeden jediný formulář. Vznikne tak n (n = počet formulářů) kompletních XML stromů. Tyto XML stromy jsou následně spojeny do jednoho výsledného stromu. Máme tak jistotu, že nepřijdeme o žádné důležité elementy, bez kterých by nebyl požadavek *<edit-config>* validní.

2.2.3 XML operace jako služba

Podobně jako třída *Data*, která je definovaná jako služba *DataModel*, také třída *XMLOperations* je definována jako služba *XMLOperations* a je dostupná pomocí DI všem kontrolerům a případným dalším třídám. Služba *XMLOperations* je přímo závislá na službě *DataModel* (platí i obráceně), protože využívá její API např. pro získání datových modelů, které dále zpracovává a provádí např. spojení odpovědi z příkazu *<get>* s datovým modelem. Všechny operace jsou napsány co možná nejobecněji tak, aby je bylo možné použít z jakéhokoliv vlastního modulu (viz. sekce Modulárnost aplikace).

2.3 Asynchronní načítání

Asynchronní načítání obsahu je důležité pro ajaxový průchod aplikací. Hlavní technikou, kterou budeme používat, je nahrazení obsahu pouze těch částí stránky, které se mají změnit. Ostatní části stránky zůstávají neměnné. Dále v textu bude tato technika nahrazení určitých částí stránky nazývána jako „invalidace bloků“.

V analýze bylo uvedeno, že vykreslená šablona celé stránky není na asynchronním načítání nijak závislá. Na stránce jsou vygenerovány odkazy na dané části aplikace, stejně tak formuláře a jejich akce. Formuláře i odkazy mohou navíc obsahovat různé modifikátory v podobě *HTML5 data atributů*, jako jsou např. modifikátory pro zákaz uložení akce do historie.

Celý skript obsluhující asynchronní načítání obsahu byl vytvořen jako jQuery plugin (soubor *jquery.netopeergui.js*).

Nyní si popíšeme průběh invalidace bloků v jeho jednotlivých částech.

2.3.1 Odchycení událostí

Prvním krokem pro invalidaci bloků jsou události kliknutí na odkaz nebo odeslání formuláře. Tyto události jsou odchyceny v JS a je zakázáno jejich výchozí zpracování. Místo toho v JS zjistíme cílovou URL, na kterou požadavek směřuje, a načteme ji pomocí ajaxového volání (za využití funkcí knihovny jQuery). Pokud uživatel odesílá formulář, připojíme navíc jako parametry serializovaná data odesílaného formuláře.

Ajaxové volání cílové URL je zpracováno konkrétní akcí v PHP (k určení, která akce se má provést slouží technika *routování*). Logika celé akce je provedena až do konce úplně stejným způsobem, jako kdyby docházelo k jejímu přímému volání. Rozdíl je ovšem ve formátu odpovědi z akce kontroleru. Standardně se jako odpověď vrací pole proměnných, které je na nižších vrstvách frameworku předáno šabloně (definována v anotacích akce kontroleru). Šablona se zpracuje a její výsledný kód je vykreslen prohlížečem. Pro invalidaci bloků ale nechceme vykreslovat celou šablonu, pouze její části, neboli bloky.

2.3.2 Definice invalidovaných bloků

Šablonovací systém TWIG využívá konstrukty zvané bloky (element *{% block %}*). Ty unikátně pojmenovávají a ohraničují části šablony, které můžeme např. v hierarchii vnořených šablon různým způsobem přetěžovat, rozšiřovat a modifikovat (vždy pouze směrem k rodičovské šabloně). Blok si můžeme představit jako *protected* proměnnou rodičovské třídy, kterou dědí několik podtříd. Šablona může být instance jakékoliv z hierarchie tříd, která mění hodnotu *protected* proměnné.

Pro invalidaci bloků chceme načítat pouze hodnoty předem definovaných bloků z jakékoliv šablony. Pro určení, který blok chceme načíst, používáme jmenný prostor šablony, ve kterém se nachází unikátní název bloku. Těchto bloků může být v odpovědi *0..n*.

V metodě *BaseControlleru*, která připravuje pole proměnných pro výpis do šablony, byla doplněna kontrola, jestli je požadavek typu *XMLHttpRequest*. Pokud ano, nedochází k vrácení pole proměnných pro šablonu a jejímu vykreslení, nýbrž k vykreslení jednotlivých bloků z definovaných šablon. Vykreslený HTML kód bloků je uložen do pole, které je serializováno jako JSON řetězec a navráceno jako odpověď.

Logika akce kontroleru, který požadavek zpracovává, musí být rozšířena pouze o definice jmenných prostorů šablon a názvů bloků, které se mají v případě ajaxového volání vložit do odpovědi. V *BaseControlleru* byly také přidány potřebné metody pro definování těchto bloků a práci s nimi.

2.3.3 Zpracování odpovědi v JS

Jako odpověď ajaxového volání získáme pole vykreslených bloků. Nyní musíme tímto novým vykresleným kódem nahradit příslušné části stránky. Stránka

nemá ve svém HTML kódu žádné informace o blocích, které byly použity v TWIG šablonách. Proto je obsah každého TWIG bloku obalen elementem, který má atribut *id = "block--NAZEV_TWIG_BLOKU"*. Díky tomu je možné nahradit z prostředí JS obsah elementu s hodnotou atributu *id* odpovídajícímu názvu obdrženého bloku novým kódem. To provedeme postupně pro všechny vykreslené bloky z odpovědi.

2.3.4 History API

Poté, co úspěšně zpracujeme všechny bloky odpovědi, musíme přidat novou URL do historie prohlížení. K tomu využijeme *HTML5 History API* a jeho metodu *pushState()*. Tím dojde ke změně URL adresy prohlížeče a také k možnosti vrátit se v prohlížeči zpět na předchozí stránku.

Pokud se uživatel pohybuje v prohlížeči zpět nebo vpřed, došlo by nyní ke klasickému načtení celé stránky. Pro ajaxové načítání musíme tuto událost odchytit obdobně jako událost kliknutí na odkaz nebo odeslání formuláře. K tomu slouží událost *window.onpopstate*. Zpracování události *window.onpopstate* proběhne stejně jako u události kliknutí na odkaz s danou URL, nedojde pouze k přidání URL do historie prohlížeče pomocí metody *pushState*. Stránka by se tam jinak objevila dvakrát.

2.4 Rozšíření možností konfigurace zařízení

Rozšíření možností konfigurace zařízení bych na začátku rozdělil do tří kategorií:

1. modifikace stromu v UI pomocí JS
2. přidání potřebné logiky pro zpracování dat z UI
3. přidání funkčnosti definované protokolem NETCONF

2.4.1 Konfigurace v UI pomocí JS

Případy užití, ke kterým v průběhu konfigurace zařízení z pohledu uživatelského rozhraní dochází, byly podrobně popsány v analýze (sekce 1.4.3). Pro shrnutí, hlavní oblasti jsou:

- editace hodnot stromu, který jsem obdržel ze serveru,
- rozšíření stromu o nové uzly či podstromy, uživatelské řazení seznamů, mazání uzlů,
- automatické našeptávání názvů uzlů,
- automatické vygenerování povinných potomků,

- přepínání mezi datovými úložišti,
- práce s prázdnými moduly a datovými úložišti,
- volání RPC metod,
- akce „commit“ (odešli všechny úpravy).

U jednotlivých modifikací stromu musí uživatel vždy provést podobné akce. Akce jsou reálné kroky, které uživatel provede v UI. Jednotlivé kroky je možné popsat následující osnovou:

1. vyvolat akci (např. kliknutím na ikonku)
2. zobrazit formulář (nejčastěji v modálním okně se zatemněním ostatních částí stránky)
3. přidat nový podstrom, editovat hodnoty uzlů
4. podívat se do nápovědy, přečíst si popis uzlů
5. kliknout na tlačítko odeslat změny

Určité DOM prvky, které reprezentují vykreslení stromu hodnot a příslušných ovládacích prvků, mají pomocí JS přiřazeny *listenery* pro odchycení určitých událostí (nejčastěji kliknutí, hover apod.). Všechny funkce, které tyto události obsluhují, jsou kvůli modulárnímu rozšíření aplikace vyčleněny do samostatného souboru, který je obsažen v *ModuleDefaultBundle*.

Generování nových DOM elementů probíhá dvojím způsobem — buď manuálně v JS nebo dynamicky pomocí asynchronního volání připravených akcí *AjaxControlleru*. Jako příklad pro manuální vytváření uvedu přidání nového uzlu. Řádek s tímto uzlem obsahuje pouze *<input/>* pro zadání názvu uzlu a pro zadání hodnoty uzlu. Druhým příkladem je vytvoření potřebných HTML elementů pro zobrazení formuláře, ovládacích prvků, překryvu obrazovky poloprůhledným pozadím apod. Obecně se jedná o generování HTML elementů, které jsou statické, neměnné.

Generování a vkládání dynamických dat slouží např. pro doplnění sémantických informací k nově vloženému uzlu, doplnění výchozích hodnot formulářových prvků apod. Zde hovoříme o generování HTML elementů pomocí šablony a jejich následném vložení do DOMu. Druhým typem asynchronně načítaných dat jsou např. hodnoty pro automatické našeptávání názvu uzlů, které se nevkládají přímo do DOMu, ale dále se zpracovávají.

Celkově je manipulace se stromem kombinací asynchronního načítání, invalidace bloků a rozsáhlé práce s DOMem.

2.4.2 Doplnění logiky pro zpracování dat

Pro nové způsoby konfigurace je nutné přizpůsobit také kontrolery. Úvod do problematiky zpracování XML souboru v *NetopeerGUI* jsem popsal již v sekci 2.2.

Logiku zpracování dat z formulářů bylo nutné rozšířit o veškeré nové způsoby konfigurace, jako jsou např.:

1. vytváření nových uzlů a podstromů místo pouhé úpravy hodnot,
2. zpracování mazání uzlů,
3. zpracování všech modifikací stromu pomocí operace „commit“,
4. vytvoření a zpracování formulářů pro vyplňování prázdných datových úložišť,
5. vytvoření a zpracování formulářů pro změnu datových úložišť, kopírování mezi nimi,
6. vytvoření a zpracování formulářů pro volání RPC metod.

Kontrolery pro tyto akce se starají pouze o přípravu konfiguračních dat, veškeré další zpracování probíhá ve službě *XMLOperations*. U zpracování formulářů a jejich transformace do XML souborů bylo nutné provést nejvýraznější refaktorování kódu.

V aplikaci *v1.0* byl celý tento kód obsažen přímo v kontrolerech, případně ve třídě *Data* (služba *DataModel*). Kód pro zpracování byl roztříštěn na několika místech, mnohdy zbytečně duplikován. Nyní je celá logika rozčleněna do menších funkčních celků, což přineslo mj. větší přehlednost, udržitelnost a také lepší testovatelnost kódu. Každá třída se stará pouze o to, k čemu je určena, což bylo nutné provést také z důvodů přípravy API pro modulární rozšíření celé aplikace.

2.4.3 Rozšíření funkčnosti protokolu NETCONF

Oproti aplikace *v1.0* byla do verze *v2.0* doplněna také podpora nových datových typů konfiguračních uzlů, jako jsou *choice* či *leafref*. Pokud datový model obsahuje tyto datové typy, stávající způsob výpisu stromu i jeho zpracování selhává, protože jejich obsah se zpracovává a vykresluje jiným způsobem než u ostatních typů.

Podle definice v [3] slouží *leafref* jako reference na konkrétní instanci typu *leaf* v datovém stromě. To znamená, že nemůžeme pracovat přímo s uzlem typu *leafref*, nýbrž musíme načíst správnou hodnotu z jeho reference určené hodnotou *path*.

Typ *choice* umožňuje podle [3] „oddělit navzájem nekompatibilní uzly do nezávislých voleb pomocí *choice* a *case* výrazu. *Choice* obsahuje seznam

case uzlů a definuje tak uzly, které se nesmí objevit v daném podstromě společně.“ Laicky řečeno se jedná o výběr jednoho podstromu uzlů z předdefinovaných možností.

Pro tento typ bylo nutné připravit speciální logiku pro výpis i zpracování. Datový model obsahuje navíc elementy *case*, které se dále v konfiguraci nevyskytují. Tudíž mapování odpovědi serveru na datový model muselo být upraveno tak, aby procházelo přes elementy *case* k jeho potomkům. Procházení k potomkům muselo být zavedeno i pro načítání hodnot pro našeptávání názvů elementů, protože element *case* se nevyskytuje ani v *<edit-config>* požadavku.

2.4.4 Výpis NETCONF notifikací

Zvláštní částí rozšíření možností konfigurace jsou NETCONF notifikace. Notifikace jsou implementovány pomocí WebSocktes (dále WS). Správné zobrazení a chování notifikací uvnitř aplikace závisí na asynchronním načítání obsahu (invalidaci bloků stránky) tak, aby nedocházelo k vícenásobnému připojování se k WS pro příjem notifikací. V dalším popisu realizace počítáme s tím, že je asynchronní načítání obsahu použito.

Implementaci komunikace pomocí WS uvnitř *NetopeerGUI* bylo provedeno v souboru *notifications.js*. Zde byl vytvořen jQuery plugin s názvem *notifWebSocket*, který se stará o veškerou komunikaci i vykreslování.

Prvním úkolem bylo určit, kde se notifikace budou na frontendu uživateli zobrazovat. Rozhodl jsem se přidat do spodní části obrazovky panel, do kterého budu notifikace automaticky přidávat. Panel vzhledem připomíná terminálové okno a je jasně odlišitelný od zbytku konfigurační části obrazovky.

Po prvním vykreslení tohoto panelu v DOMu (zpravidla po zobrazení první konfigurační stránky zařízení) se aplikace připojí k definovanému portu WS — naváže komunikaci se serverem. Pokud je již spojení navázáno (ajaxový průchod aplikací), nové se nevytváří.

Plugin *notifWebSocket* registruje *listenery* na potřebné události definované v HTML5 standardu WebSockets:

- *onopen*: vypíše na obrazovku zprávu, že spojení bylo navázáno
- *onclose*: vypíše na obrazovku zprávu, že spojení bylo ukončeno
- *onmessage*: vypíše na obrazovku obdrženou notifikaci
- *onerror*: vypíše na obrazovku chybu

Jakýkoliv výpis na obrazovku doprovází vizuální probliknutí panelu. Každá vypsaná zpráva obsahuje také přesný čas, kdy byla zpráva vytvořena. Jednotlivé stavy (informační, chybová, úspěšná) zprávy jsou barevně odlišeny.

Notifikace obsahují v některých případech také strukturovaná data, např. informace o IP adrese, ze které se připojil jiný uživatel k aktivnímu zařízení.

Tato strukturovaná data nejsou vypisována na obrazovku jako obyčejný text, ale jsou obohacena o určité chování — např. kliknutím na IP adresu se zobrazí dostupné informace o přibližné poloze, hostname a další.

2.5 Modulárnost aplikace

V úvodní analýze jsem došel k závěru, že modulárnost aplikace bude realizována pomocí samostatných Symfony2 Bundle. Realizaci modulárního rozšíření popíšu postupně od nejnižší logiky aplikace pro výběr aktivního *bundle*, přípravu API, přes způsoby implementace vlastního rozšíření z *bundle* až po reálné příklady použití.

2.5.1 Logika pro výběr aktivního *bundle*

V analýze bylo rozhodnuto, že je nutné v databázi připravit mapovací tabulku *jmenného prostoru modulu* na *jmenný prostor bundle*. Ta byla nadefinována v entitě *ModuleController*.

Pro výběr *bundle*, který má být vypsán, byla vytvořena služba *ModuleListener*, obsahující metodu *onKernelController*, ve které je řízena logika rozhodování. Ukázka kódu této metody je uvedena na výpise 2.5. Kromě automatického výběru prvního kontroleru je ukládán naposledy zvolený kontroler do struktury *ConnectionSession*, aby bylo možné uživateli při příštím přístupu automaticky zobrazit správný vzhled modulu.

2.5.2 Definice *ModuleControllerInterface* a příprava API

ModuleControllerInterface definuje jedinou povinnou metodu *moduleAction()*, která řídí vykreslování veškerých konfiguračních dat zařízení. V analýze byl popsán způsob, jakým má být vytvořen obecný *ModuleController*, který nabízí metody pro přípravu a zpracování konfiguračních dat a také ukázku komunikace se službou *DataModel* pro komunikaci s NETCONF serverem. *ModuleController* vznikl vyčleněním logiky pro přípravu konfiguračních dat z původního *DefaultControlleru*, který nyní slouží pouze pro vykreslování těch částí aplikace, které se netýkají konfigurace modulu.

ModuleController obsahuje metodu *prepareDataForModuleAction()*, která vrací zpracovanou odpověď ze serveru obohacenou o informace datového modelu. Tato odpověď je také předána šabloně a může být dále zpracována dle potřeby. Metoda připravuje rozšiřujícímu *bundle* veškerá potřebná data pro další zpracování a je tedy vhodné, nikoliv nezbytné, aby ji rozšiřující *bundle* používal.

Obecné API pro zpracování dat a komunikaci s NETCONF serverem bylo již popsáno v předchozích kapitolách. Pro shrnutí uvedu, že pro komunikaci s NETCONF serverem slouží služba *DataModel*, pro zpracování XML souborů, datových modelů apod. pak služba *XMLOperations*. Tyto služby jsou dostupné

```
public function onKernelController(GetResponseEvent ↙
    ↳ $event) {
 $dm = $this->dataModel;

 $attributes = $event->getRequest()->attributes;
 if (in_array($attributes->get("_route"),
                array("module", "subsection"))) {

  // get available namespaces for this connection
  $namespace = $dm->getNamespaceForModule(...);
  if ($namespace !== false) {
   $record = $dm->getModuleControllers(...);

   if ($record) {
    // get all saved controllers from DB
    $controllers = $record->getControllerActions();

    $conn = $dm->getConnectionSessionForKey(...);
    $activeController = $conn->↙
        ↳ getActiveControllersForNS(...);

    // if we don't have any saved (preferred)
    // controller, we will use the first one from DB
    if (!$activeController) {
     $activeController = $controllers[0];
    }

    $event->getRequest()->attributes->set("_controller↙
        ↳ ", $activeController);
   }
  }
 }
}
```

Výpis 2.5: Implementace *onKernelController()* pro výběr konkrétního *bundle*.

odkudkoliv z kontroleru přes *ServiceContainer* nebo pro další vlastní služby díky *Dependency Injection*.

2.5.3 Rozdělení *assets*, vytvoření jQuery pluginů

Nejen PHP třídy a kontrolery musely projít velkou proměnou. Pro implementaci modulárního rozšíření bylo nutné rozdělit také všechny *assets* a realizovat

jejich automatické načítání (podle zvoleného aktivního vzhledu modulu).

Jako první byly vyčleněny všechny SASS proměnné do samostatného souboru a přepsány fixně nadefinované vlastnosti z CSS souborů do proměnných. Vznikl tak konfigurační soubor všech proměnných prostředí, které je možné libovolně upravovat nebo přetěžovat tak, aby bylo možné přizpůsobit vzhled aplikace podle individuálních požadavků. Obecný vzhled i layout aplikace, stejně tak výchozí *bundle* tyto proměnné využívají.

Z obecných JS aplikace byl vyčleněn kód patřící *ModuleDefaultBundle*. Ostatní JS kód byl zpřehledněn, refaktorován, přesunut do samostatných souborů nebo z něj byly vytvořeny nové jQuery pluginy.

Poznámka: Na konci akce ajaxového průchodu se vždy volá funkce *initModuleDefault()* (pokud existuje). Toto je vstupní funkce pro provedení logiky definované v aktuálně načteném *bundle*.

2.5.4 Vytvoření ukázkových modulárních rozšíření

Hlavním *bundle* pro zobrazení konfiguračních informací je výchozí *ModuleDefaultBundle*. Všechny rozšiřující *bundle* jsou uloženy pod jmenným prostorem $FIT\backslash Bundle$ místo $FIT\backslash NetopeerBundle$, kde je uložen samotný kód aplikace. Základní aplikace *v2.0* nabízí celkem 2 *bundle*, zmíněný *ModuleDefaultBundle* a *ModuleXMLBundle*, který nabízí pouze „obarvený“ výpis XML stromu na stránku bez jakékoliv další logiky.

V rámci vývoje nových rozšíření jsem vytvořil pro dva studenty bakalářského studia dva nezávislé *bundle*, *ModuleNemeaBundle*[32] a *ModuleLinuxBundle*[33]. Studenti by v rámci svých bakalářských prací měli doplnit rozšíření *NetopeerGUI* o interaktivní prezentaci modulu NEMEA (*ModuleNemeaBundle*) a také vylepšenou konfiguraci linuxového stroje (*ModuleLinuxBundle*).

U těchto dvou připravených *bundle* jsem popsal také návod na instalaci a povolení daného modulu v aplikaci. Instalace není nijak složitá a využívá standardní nástroje pro správu závislostí v PHP *composer* a obecné postupy pro instalaci *Symfony2 Bundle* do Symfony projektu (více o instalaci v sekci 2.9).

2.5.5 Jakým způsobem lze využít rozšíření v praxi

Pro vykreslení jakékoliv konfigurační stránky je použita jedna obecná metoda *ModuleAction*. Programátor má ve svém rozšiřujícím *bundle* naprostou volnost a metodu může implementovat podle vlastní potřeby. Může např. využít API a připravené metody pro vlastní způsob výpisu konfiguračních dat, nebo také tuto funkcionalitu úplně obejít a naimplementovat do obsahové části konfigurace zařízení celý vlastní vzhled bez aplikací definovaného layoutu a využívat

[32] `https://github.com/CESNET/Netopeer-GUI-module-nemea`
[33] `https://github.com/CESNET/Netopeer-GUI-module-linux`

tak pouze *backend* aplikace. Toto chování dává smysl pro režim jednoho zařízení (více v sekci 2.6), kdy bude chtít výrobce zobrazit třeba pouze obrázek *toasteru* a toastového chleba, doplnit ho vhodnou animací jako ukázku volání RPC metod a obrovské univerzálnosti protokolu NETCONF.

Jiným příkladem může být pouhá změna vzhledu jednoho modulu, např. u ukázkového *bundle* pro správu linuxového serveru. Nastíním zde představu, jak by toto rozšíření mohlo vypadat. Vycházejme z toho, že na linuxovém serveru je spuštěn NETCONF server a nainstalován LINUX modul pro konfiguraci linuxu přes protokol NETCONF. Představme si konfiguraci síťových rozhraní. Ty můžeme nyní konfigurovat buď z prostředí příkazové řádky, pomocí výchozího systémového GUI, nebo přes *NetopeerGUI* a jeho výchozí vzhled konfigurace. Rozšíření by mohlo přinést alternativu grafické reprezentace konfiguračních dat v podobě „krabiček“, které bude možné propojovat „dráty“, různě řadit, přidávat atd. Fantazii se meze nekladou.

2.6 Režim jednoho zařízení

Pro určení, zda bude aplikace spuštěna v režimu jednoho zařízení, je možné využít vlastních proměnných pro konfiguraci aplikace uvedených v souboru *app/config/parameters.yml*. V průběhu instalace (nebo také později manuálně) je možné nastavit hodnotu nového parametru *netopeer_single_instance* na hodnoty *true/false*. V souboru *app/config/config.yml* jsou pak nastaveny hodnoty IP adresy zařízení a portu (výchozí *localhost:830*), ke kterému se v případě režimu jednoho zařízení aplikace připojí.

Pokud je hodnota parametru *netopeer_single_instance* nastavena na *true*, zobrazí se automaticky místo přihlašovacího formuláře rovnou formulář pro zadání jména a hesla pro přímé připojení se k zařízení. Po úspěšném připojení se uživateli zobrazí rovnou detail zařízení, které může začít volně konfigurovat.

Aplikace má již od své verze *v1.0* povinné přihlašování uživatelů, aby bylo možné identifikovat uživatele (např. kvůli historii připojených zařízení atp.). Všechny stránky aplikace kromě přihlašovací obrazovky vyžadují roli přihlášeného uživatele. Je tedy nutné, aby došlo k autentizaci uživatele také v režimu jednoho zařízení.

Pro tento speciální případ byl vytvořen uživatel s uživatelským jménem *localhostUser*, kterého je po úspěšném připojení se k zařízení automaticky přihlášen na pozadí. Pro automatické přihlášení bylo nutné rozšířit *SecurityController* o akci *singleInstanceLoginAction()*. Bylo také nutné správně odchytávat a předávat informace o neúspěšném připojení se k zařízení.

2.7 Přihlašování pomocí SAML

Security Assertion Markup Language (SAML) je podle [25] standard založený na XML poskytující mechanismus pro výměnu autentizačních a autorizačních

dat mezi zúčastněnými stranami, tj. poskytovatelem služeb a poskytovatelem identity. V praxi řeší SAML problém jednotného přihlašování na více webů — Single Sign-On (SSO). V případě *NetopeerGUI* je cílem používat SAML pro přihlášení uživatelů pomocí EduID[34].

Pro implementaci přihlašování pomocí SAML byl použit volně dostupný SamlSPBundle[35], který nabízí implementaci celého procesu autentizace pomocí SAML.

Prvním krokem bylo nastavení potřebných parametrů samotného *bundle*. Základní konfigurace byla provedena podle návodu uvedeného v manuálu SamlSPBundle. Pro chování aplikace je potřeba mít jednoznačně identifikovaného uživatele proto, aby bylo možné si ukládat historii připojených zařízení apod. Proto musíme zařídit také jednoznačnou identifikaci uživatele přihlašujícího se pomocí EduID. Musel jsem proto implementovat poněkud složitější logiku pro ukládání a identifikaci přihlášeného uživatele pomocí vlastního *User Providera*.

Informace o uživateli, které obdržím jako odpověď po přihlášení pomocí SAML, není možné jednoduše ukládat do stávající databázové struktury uživatelů. Vytvořil jsem proto novou entitu *SamlUser*. Z původní tabulky uživatelů byla vyčleněna společná data potřebná pro oba dva typy uživatelů do tabulky *userData* (definice role uživatele a jeho individuální nastavení).

V další fázi bylo nutné doplnit logiku přihlašování do aplikace tak, aby bylo možné přihlásit se nezávisle buď pomocí přihlašovacích údajů uložených v lokální databázi (původní tabulka *User*), nebo pomocí EduID (nová tabulka *SamlUser*) nebo pomocí uživatele definovaného v konfiguračních souborech (*in_memory*).

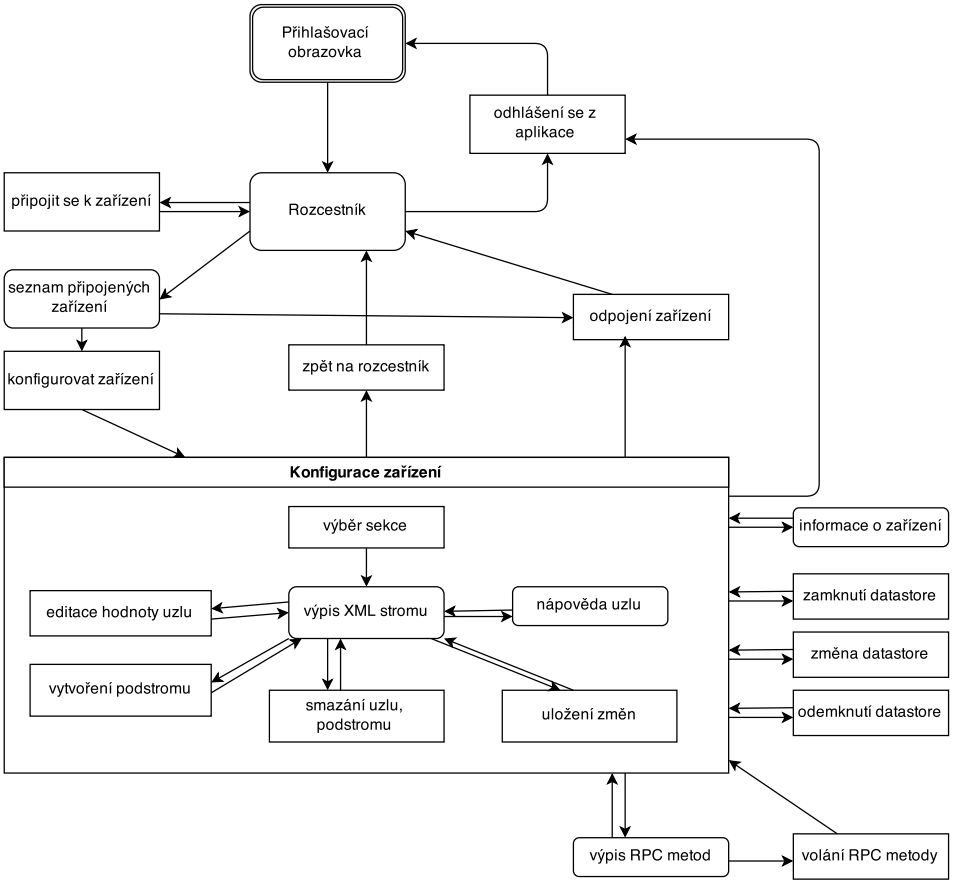
2.8 Uživatelské rozhraní

Původní vzhled aplikace *v1.0* vycházel z „drátových modelů“ (dále *wireframe*) jednotlivých typových stránek. Tyto *wireframy* jsem pro aplikaci *v2.0* rozšířil o nové možnosti konfigurace. Nástrojem, který jsem pro tvorbu wireframů použil, je online služba MockFlow[36]. Výsledné náhledy jsou obsaženy v příloze D.1. Wireframy byly také testovány v rámci projektu předmětu MI-NUR.

Kromě drátových modelů jsem pro představu o jednotlivých částech aplikace připravil tzv. „Task Graph“ (zobrazen na obrázku 2.6 a 2.7). Ten popisuje, jakým způsobem může uživatel s aplikací pracovat.

V aplikaci *v2.0* jsem se zaměřil na přívětivost a přehlednost uživatelského rozhraní. Na základě uživatelského testování i zpětné vazby uživatelů jsem změnil např. rozložení některých prvků layoutu.

[34] `https://www.eduid.cz`
[35] `https://github.com/aerialship/SamlSPBundle`
[36] `http://www.mockflow.com`



Obrázek 2.6: UI: Task Graph. Graf jednotlivých akcí a obrazovek, které jsou v aplikaci a jejího UI k dispozici. Tento graf nepopisuje situaci při použití aplikace v režimu jednoho zařízení. Graf pro režim jednoho zařízení je popsán na obrázku 2.7.

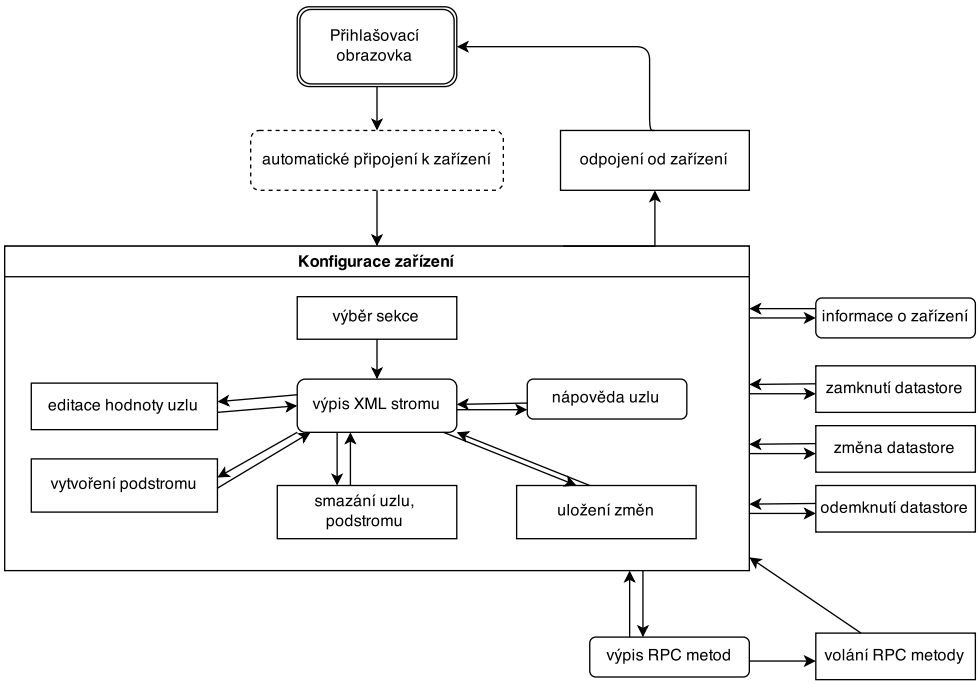
Celkově bych mohl změnu vzhledu rozdělit do dvou oblastí:

1. Grafická podoba

   - Oproti *v1.0* jsem odstranil barevné přechody a přebytečné stíny. Cílem bylo dosáhnout moderního designu, tzv. „flat designu“.
   - Projasnění a sjednocení barev (jsou definovány v jednom souboru obsahujícího proměnné používané napříč celou aplikací).

2. Layout

   - Z levého sloupce, který obsahoval většinou zbytečné prázdné místo, se stal ovládací panel pro konfiguraci zařízení. Kromě přepínání



Obrázek 2.7: UI: Task Graph - režim jednoho zařízení. Graf jednotlivých akcí a obrazovek, které jsou v aplikaci v režimu jednoho zařízení a jejího UI k dispozici.

sekcí nyní nabízí také změnu datového úložiště, volání RPC metod nebo změnu vzhledu výstupu aktivního modulu.

- Sjednocení a zarovnání ovládacích prvků pro každý řádek (uzel) stromu, nezávisle na úrovni zanoření. Nedochází tak k nechtěným překryvům pro menší rozlišení okna prohlížeče.

- Přepracování horní lišty navigace — doplnění uživatelského menu, přidání možnosti odpojení se od zařízení místo matoucího odhlášení se z aplikace.

- Přidání panelu flash zpráv umístěného v pravé části obrazovky. Tento panel je zobrazen pouze po kliknutí na ikonku v horní liště navigace.

- Přidání panelu pro výpis notifikací NETCONF serveru. Uživatel si může nastavit výšku tohoto panelu pro zobrazení více informací na úkor velikosti panelu konfigurace.

2.9 Jednoduchá instalace a distribuce

Aplikace *v1.0* byla vyvíjena zpočátku pouze interně. U aplikace *v2.0* jsme se rozhodli, že celý vývoj přesuneme z interních repozitářů na GitHub[37]. K tomuto kroku nás vedla snaha o rozšíření tohoto nástroje mezi více uživatelů a hlavně získání zpětné vazby.

Aby bylo možné aplikaci dostat mezi více uživatelů, bylo nutné zjednodušit celý proces instalace. Před instalací samotného *NetopeerGUI* je potřeba nejdříve nainstalovat systémové závislosti. *NetopeerGUI* je závislé na PHP v5.4, *mod_netconf*, některých rozšířeních pro práci s XML apod. Kompletní seznam závislostí je uveden v příloze C. *Mod_netconf* je vyvíjen ve vlastním repozitáři a proto jsem se rozhodl vložit jej do repozitáře *NetopeerGUI* jako *git submodule* do složky *install*. Všechny potřebné závislosti je možné nainstalovat ze složky *install*, návod je prezentován v README repozitáře.

Ve druhé fázi je potřeba nainstalovat samotnou aplikaci. V aplikaci *v1.0* byl celý projekt uložen v repozitáři, včetně *vendor* adresáře obsahující framework Symfony2. Správa jakýchkoliv závislostí aplikace na dalších knihovnách či *bundle* byla komplikovaná a hlavně manuální. Proto jsem se rozhodl použít systém Composer[38] pro správu závislostí v PHP. Composer používá jednoduchý JSON soubor *composer.json* pro definici závislostí a jejich verzí. Umožňuje také zadání příkazů, které se mají provést před nebo po instalaci/aktualizaci. Na tomto místě můžu nastavit další potřebné příkazy, jako je vygenerování *assets*, nastavení práv souborů apod.

Abychom uživatelům mohli usnadnit práci co nejvíce a aby si mohli vyzkoušet *NetopeerGUI* bez nutnosti instalace, rozhodli jsme se nabízet celý nainstalovaný systém včetně všech závislostí také v podobě virtuálního obrazu. Tento virtuální obraz obsahuje jedno lokální zařízení, ke kterému je možné se připojit a vyzkoušet si tak všechny možnosti konfigurace bez potřeby připojení k internetu.

2.10 Dokumentace kódu

Dokumentace kódu je důležitým zdrojem informací pro jakékoliv další programátory, ale také pro vývoj aplikace z IDE. IDE zpravidla nabízí automatické našeptávání parametrů, kontrolu datových typů apod. právě na základě informací uvedených v dokumentaci.

Důležitest dokumentace se ještě zvýšila poté, co bylo *NetopeerGUI* zveřejněno a bylo připraveno API pro modulární rozšíření. Programátorskou dokumentaci jsem doplnil pro všechny vlastní třídy a kontrolery. Zdokumentovány jsou všechny soubory, metody a funkce, včetně vstupních a výstupních parametrů. Pro vygenerování dokumentace jsem použil nástroj phpDocumenta-

[37] `https://github.com/CESNET/Netopeer-GUI`

[38] `https://getcomposer.org`

tor 2[39]. Dokumentace se, podobně jako u jiných nástrojů, píše do komentářů přímo k popisovanému objektu. Pro PHPDoc je využíván systém anotací.

Výsledná dokumentace je zobrazitelná v rámci *NetopeerGUI* na adrese */bundles/netopeer/docu/*.

[39] `http://www.phpdoc.org`

Kapitola 3

Testování

V rámci vývoje bylo nutné provádět průběžné testovaní aktuálně implementovaných funkcí, hlavně pak práci s XML stromem. Pro kontrolu správnosti výstupu PHP skriptů, které vytváří obsah zpráv posílaných protokolem NETCONF, jsem použil referenční výstup konzolové aplikace *netopeer-cli*.

Veškeré důležité akce, které jsou v *NetopeerGUI* vykonány (např. komunikace se serverem nebo načítání datových modelů), jsou logovány pomocí *služby Monolog*. Jakmile nastane po provedení určité akce chyba, je možné získat z chybového logu sekvenci příkazů, které vyústily v chybu, včetně odeslaných dat. V konzolové aplikaci provedu stejnou sekvenci příkazů a zkontroluji shodu s odpovědí serveru. Tím dokážu zjistit, jestli se jedná o chybu způsobenou v *NetopeerGUI* či o chybu nižších vrstev aplikace.

Toto ruční testování je vhodné pouze při implementaci nové funkcionality nebo ladění chyb, nejedná se o dlouhodobé řešení. Rozhodl jsem se proto využít i některé další způsoby testování aplikace pomocí těchto úrovní testů:

1. jednotkové testy,
2. akceptační testy,
3. funkcionální testy,
4. uživatelské testování.

Zdrojový kód a chování aplikace není možné rozumně otestovat pouze jedním druhem testu, protože každý druh je zpravidla vhodný pro testování určité funkcionality. Hovoříme zde proto o několika úrovních testování. Mezi těmito úrovněmi volně přecházím tak, abych byl schopen testy pokrýt co nejvíce případů užití aplikace.

Veškeré zdrojové soubory testů jsou umístěny v adresáři *src/FIT/NetopeerBundle/Tests*.

3.1 Jednotkové testy

Zdroj [26] popisuje jednotkové, neboli *unit* testy takto: „Nejnižší možná úroveň testů, kde testujeme izolovaně a samostatně jednotlivé třídy. Na této úrovni nás zajímají především správné návratové hodnoty metod v závislosti na vstupních parametrech. Při jednotkovém testování bychom se měli snažit o maximální izolaci testované jednotky, měli bychom se pokud možno vyvarovat:

- volání metod jiných skutečných tříd,
- přistupování k síti (internet, intranet),
- používání databází,
- používání souborového systému.

Pro simulaci těchto činností se používají falešné objekty, které nevykonávají žádnou skutečnou činnost, pouze splňují nějaké požadované rozhraní. Tyto objekty se označují jako *mocks* nebo *stubs*.“

Pro jednotkové testy používám standardní nástroj PHPUnit[40].

3.1.1 Okruhy pokrytí testy

Jednotkovými testy jsem pokryl většinu metod, které obsahuje služba *XMLOperations*. Pokryto je zpracování datových modelů, nahrazení hodnot stromu, přidávání podstromů, odebírání hodnot stromu, testování validace XML souboru, příprava a kompletace XML požadavku pro *<edit-config>* a všechny podpůrné metody. Vynechal jsem pouze metody obsluhující zpracování formulářů, které interně pouze předávají hodnoty *$_POST* právě těmto testovaným metodám.

Většina metod je závislá na XML souboru obsahující datový model. Načítání datového modelu ze souborového systému jsem v těchto testech musel „mockovat“. Stejně tak jsem si vytvořil „mockovací“ službu *DataModel*, kterou do instance testované třídy *XMLOperations* vkládám jako DI místo původní. Zde se projevuje velká síla návrhového vzoru DI. Pro testovací prostředí stačí změnit v konfiguračním souboru cestu ke vkládané třídě. V kódu třídy *XMLOperations* už pak není potřeba nic měnit.

Ostatní třídy již kvůli své závislosti buď na komunikaci s NETCONF serverem nebo *$_POST* hodnotách nemá smysl pomocí jednotkových testů testovat. Otestování chování zbylých částí aplikace bude provedeno následujícími úrovněmi testů.

[40] `https://phpunit.de`

3.2 Testy uživatelského rozhraní – akceptační testy

Další úrovní testů jsou akceptační testy. Zdroj [27] popisuje tyto testy následovně: „Akceptační testy mohou být prováděny jakoukoliv i netechnickou personou. Tato persona může být tester, manager nebo klidně klient. Pokud vyvíjíme webovou aplikaci, testeři nepotřebují nic více než webový prohlížeč, aby otestovali, že aplikace funguje v pořádku. Díky akceptačním testům můžeme reprodukovat akce, které provedl tester podle testovacích scénářů a spustit je automaticky po každé změně kódu.

Akceptační testy nejsou nijak závislé na CMS nebo frameworku, který aplikace používá. Nejsou závislé ani na programovacím jazyce, protože testujeme výslednou aplikaci běžící ve webovém prohlížeči.“

Pro akceptační i následující funkcionální testy jsem se rozhodl použít framework Codeception[41]. Důvodem, proč jsem si zvolil framework Codeception byla podrobná dokumentace, ale hlavně jednoduchá, přehledná a čitelná syntaxe pro psaní testů.

Codeception umožňuje spouštět akceptační testy pomocí PhpBrowser (jedná se o komponentu frameworku Symfony2) nebo pomocí Selenia. PhpBrowser pouze simuluje chování prohlížeče a neumí např. testovat JS události a aplikace založené na JS. Proto spouštím akceptační testy s využitím Selenium Webdriver[42]. Jedná se o rozšíření původního Selenia 1.0, které jsem používal již v *NetopeerGUI v1.0*. Dle oficiálního popisu [28] je Selenium „nástroj pro automatizované testování webových aplikací, ale není limitován pouze pro ně — jednoduché úkony v administračním systému mohou (a měly by) být otestovány také“.

Pro automatizované testování pomocí Selenia je nutné mít spuštěný *Selenium server*. Jedná se o Java aplikaci, která je volně dostupná na adrese [43].

Poznámka: Codeception umožňuje také psaní jednotkových testů. Jednotkové testy jsem měl ale v době psaní již naimplementované pomocí PHPUnit.

3.2.1 Okruhy pokrytí testy

Okruhy pokrytí jsem se snažil co nejvíce přiblížit testovacím scénářům připravených pro uživatelské testování (více v sekci 3.4):

1. přihlášení do systému a připojení k zařízení,
2. úprava, vytvoření a smazání elementu,
3. úprava dat v datovém úložišti *candidate* a následné nahrání do datového úložiště *running*,

[41] `http://codeception.com`
[42] `http://www.seleniumhq.org/projects/webdriver/`
[43] `http://docs.seleniumhq.org/download/`

4. přepnutí mezi připojenými zařízeními a následné odhlášení,
5. zapnutí a vypnutí modulu,
6. vytvoření nového kořenového elementu,
7. volání RPC metody.

Akceptační testy testují chování celé aplikace, tedy také funkčnost komunikace s NETCONF serverem, správné chování *routování* systému pro zobrazení definovaných stránek, správné chování ajaxového průchodu aplikací, JS události pro práci s XML stromem a celkovou funkčnost celé aplikace. Chyby nalezené v akceptačních testech dokáží odhalit chybné chování jakékoliv části aplikace (samozřejmě nenalezne všechny chyby, pouze ty, které mohou nastat v průběhu testu). Jedná se o nejuniverzálnější typ testu.

3.3 Funkcionální testy

Na úrovni mezi jednotkovými a akceptačními testy leží funkcionální testy. Zdroj [29] popisuje funkcionální testy takto: „Poté, co jsme napsali akceptační testy, funkcionální testy jsou téměř totožné, ale s jedním hlavním rozdílem — funkcionální testy nevyžadují spuštěný web server.

Jednoduše nastavíme hodnoty $_REQUEST, $_GET a $_POST proměnných a zavoláme akce kontrolerů z prostředí testů. Tyto testy mohou být přínosné hlavně díky své rychlosti a také díky tomu, že nabízí často detailní výpis chyby při pádu.“

Pro funkcionální testy používám, stejně jako pro akceptační, framework Codeception.

3.3.1 Okruhy pokrytí testy

U popisu jednotkových testů jsem uvedl, že pomocí nich není jednoduché testovat např. zpracování formulářů. V našem případě jsou právě pro otestování metod pro zpracování formulářů vhodné funkcionální testy. Je pravda, že zpracování formulářů bylo již otestováno pomocí jednotlivých akceptačních testů — ale pouze globálně.

Při vývoji je ale mnohem rychlejší spouštět testy z příkazové řádky, místo zdlouhavého testování pomocí prohlížeče. Proto jsem se rozhodl otestovat alespoň základní formuláře pro konfiguraci zařízení, které v *NetopeerGUI* používám. Jmenovitě to jsou formuláře pro:

1. editaci hodnot stromu,
2. vložení nového podstromu,
3. smazání uzlu včetně jeho potomků,

4. vytvoření nového kořenového elementu.

3.4 Uživatelské testování

3.4.1 Heuristické vyhodnocení

V rámci předmětu MI-NUR (Návrh uživatelského rozhraní) jsem *NetopeerGUI* otestoval metodou heuristického vyhodnocení. Jako experty pro vyhodnocení tohoto průchodu jsem zvolil sebe a další dva spolupracovníky v mém týmu.

Oproti kompletního výčtu otázek Nielsenovy heuristiky[44] jsem seznam mírně upravil pro tuto konkrétní aplikaci. Níže uvádím seznam otázek a jejich vyhodnocení (sjednocené závěry všech expertů). V případě nalezení závady byla závada označena na stupnici závažnosti jako nízká, střední, vysoká.

1. Viditelnost stavu systému — systém by měl vždy dát uživateli vědět, co se právě odehrává.

   Vyhodnocení: Stav aplikace je viditelný v horním menu nebo podmenu označením aktivní položky (kde se uživatel nachází).

2. Poskytuje každá akce zpětnou vazbu srozumitelnou pro uživatele?

   Vyhodnocení: Systémové hlášky nalezneme v záložce vpravo nahoře. Podbarvují se zeleně nebo červeně podle typu hlášky.

3. Jsou názvy, popisky a jinak používané termíny srozumitelné cílové skupině? Jsou názvy kategorií (například v menu) voleny s ohledem na srozumitelnost pro cílovou skupinu?

   Vyhodnocení: Názvy a popisky vyjadřují přesně to, co by člověk očekával. Cílové skupině jsou srozumitelné.

4. Může se uživatel dostat z každé situace? Obejde se takové opuštění bez nutnosti opakovat dlouhou sekvenci akcí?

   Vyhodnocení: Uživatel se dostane z každé situace, ovšem při úpravě uzlů XML stromu aplikace nenabízí akce *undo* a *redo*. Po odeslání změn tak není možnost návratu zpět k předchozí verzi konfigurace. Tento problém je nutné řešit na nižších vrstvách aplikace, proto není možné závadu odstranit hned. Dále je tento problém diskutován v sekci 3.4.3. *Závažnost problému střední.*

5. Jsou všechny objekty, akce a jiné prvky vidět kdykoliv je potřeba, aniž by si je uživatel musel pamatovat?

   Vyhodnocení: Vše je dobře viditelné.

[44] `http://www.usability.gov/methods/test_refine/heuristic.html`

6. Konzistence a standardizace – uživatelé by neměli být nuceni přemýšlet, jestli různé termíny znamenají to stejné, proto se doporučuje dodržovat obecné zásady.

   Vyhodnocení: UI dodržuje obecné zásady. Nikdo nenarazil na nejednoznačnost nějakého prvku.

7. Prevence chyb – vyvarovat se chybovým hlášením bezpečným designem, který bude preventivně působit proti problémům.

   Vyhodnocení: Aplikace předchází problémům validací formulářů a našeptáváním při vyplňování polí. Při mazání části podstromu je uživateli zobrazeno potvrzovací okno. Při pokusu o přechod na jinou stránku je prováděna kontrola změn formulářů — pokud byl formulář změněn, uživateli se zobrazí oznamovací okno.

8. Estetický a minimalistický design – bez nepotřebných informací.

   Vyhodnocení: V aplikaci se nevyskytují nepotřebné informace.

9. Pomoc uživatelům poznat, pochopit a vzpamatovat se z chyb – chybové hlášky by měly být uváděny v přirozeném jazyce a měly by navrhovat řešení.

   Vyhodnocení: Chybové hlášky jsou srozumitelné. Úspěšné hlášky se ovšem rovnou schovávají pod tlačítko menu, kde se zobrazí až po kliknutí na něj. Mohly by se zobrazovat rovnou do vyskakovacích oken, alespoň na malou chvíli, aby mohl uživatel rozhodnout o jejich důležitosti. *Závažnost nízká.*

3.4.2 Uživatelské testování na základě testovacích scénářů

Pro potřeby testu bylo vytvořeno 8 podrobných testovacích scénářů, které byly předloženy testovaným subjektům. Testovací scénáře jsou dostupné na přiloženém na CD nebo v příloze E.

Testování se zúčastnilo celkem 5 testerů. Testování probíhalo u prvních třech testerů kvůli rozdílné geografické poloze vzdáleně. Testování se zúčastnili.

- Bc. Marek Švepeš: muž, vývojář modulů pro NETCONF
- Bc. Zdeněk Rosa: muž, detekce anomálií sítí v rámci CESNET
- Erik Šabík: muž, vývojář modulů pro CESNET
- Bc. Tomáš Benák: muž, student ČVUT FIT
- Bc. Marek Manukjan: muž, student ČVUT FIT

Všem testovaných subjektům byl před a po testu předložen dotazník. Jejich znění a výsledky jsou uvedeny v příloze E.

Testování objevilo následující chyby:

- Pokud už byl uživatel dříve přihlášen, poté automaticky odhlášen (nebo kliknul na tlačítko odhlášení), zobrazí se mu část detailu zařízení místo rozcestníku (identifikace problému: chyba načítání obsahu pomocí AJAX)
- Chyba v datovém modelu pro automatické našeptávání hodnot
- Při úpravě hodnoty elementu uzlu se hodnota nezměnila, nýbrž zduplikovala (problém: pravděpodobně chybně vygenerovaný edit-config)

3.4.3 Závěr uživatelského testování

Některé chyby nalezené v testech jsou problémem konfigurovaných zařízení, což z GUI nelze nijak ovlivnit. Je jasné, že by celé GUI mělo dobře reagovat na chybové stavy. Snahou je zobrazit konkrétní informaci o tom, že zařízení, které uživatel konfiguroval, např. přestalo odpovídat. Důležitá bude implementace *undo* a *redo* funkce a předvalidace formulářů za běhu, která vypíše uživateli chybové hlášení před jeho samotným odesláním.

Způsob testování na dálku není podle mě vhodný, nalezené chyby nejsou podle mého názoru úplné a dostačující. Není možné zjistit přímou příčinu selhnání. Vhodnější je forma testování s pozorováním testovaného subjektu.

Závěr

Jsem rád, že vývoj původní aplikace *NetopeerGUI* nebyl s odevzdáním bakalářské práce ukončen a pokračuje stále dále. Dlouhodobá forma vývoje mi umožnila posunout možnosti aplikace o velký krok kupředu. Diplomová práce mi v praxi umožnila uplatnit výraznou část znalostí nabytých během studia. Měl jsem možnost vyzkoušet si nové technologie a postupy, které se mi podařilo úspěšně využít také u jiných vlastních projektů.

Velkou zkušeností byla možnost být součástí rozsáhlého projektu, který byl realizován ve spolupráci s CESNETem.

Shrnutí průběhu a výsledků práce

Diplomová práce navazuje na vývoj aplikace *NetopeerGUI* provedený v rámci bakalářské práce. Díky tomu jsem již měl potřebné znalosti protokolu NETCONF a jeho využití. Mohl jsem se proto naplno věnovat vývoji vylepšení a rozšíření původní aplikace *v1.0*.

Na začátku bylo nutné získat zpětnou vazbu uživatelů, kteří aplikaci *v1.0* používali. Díky této zpětné vazbě vznikl základní seznam požadavků na rozšíření aplikace. Tyto požadavky jsem volně popsal pomocí případů užití. K případům užití jsem v analýze navrhnul možná řešení. Z návrhů možných řešení pak vznikl seznam požadavků na novou verzi aplikace *v2.0*.

Implementace nových funkcí si žádala přerozdělení funkčních bloků aplikace — tříd a kontrolerů — do menších, samostatných celků. Bez rozdělení zdrojových souborů spojeného s refaktorováním kódu by nebylo možné jednoduše implementovat modulární rozšíření. Hlavními přínosy rozdělení zdrojových kódů je větší přehlednost, jednoduchost a testovatelnost kódu. Díky rozšíření možností konfigurace zařízení je nyní možné konfigurovat i prázdný modul a také libovolně upravovat a rozšiřovat strom konfigurací. Modulární rozšíření aplikace otevírá nové možnosti dalším vývojářům pro aplikování individuálních požadavků na vzhled konkrétního modulu.

Díky testování několika úrovněmi testů se mi podařilo odstranit velké množství chyb. Lepších výsledků dosahoval také vývoj nových rozšíření. Pro aplikační a funkcionální testy jsem použil framework Codeception, pro jednotkové testy pak PHPUnit. Frontend aplikace byl podroben také rozsáhlému uživatelskému testování, které přineslo cennou zpětnou vazbu.

Má práce přináší vylepšené grafické webové uživatelské rozhraní pro správu a konfiguraci zařízení komunikujících pomocí protokolu NETCONF.

Veškeré požadavky, které byly na základě zpětné vazby uživatelů v zadání definovány, byly splněny.

Budoucí práce

V této sekci shrnu plánované pokračování vývoje *NetopeerGUI* nad rámec původních cílů. Věřím, že vývoj aplikace po odevzdání diplomové práce, podobně jako po odevzdání bakalářské práce, neskončí a bude dále pokračovat.

Nejbližší vývoj aplikace bude zaměřen na vylepšení chování uživatelského rozhraní z pohledu uživatele hlavně o:

1. obnovu stavu zařízení nahráním zálohy konfigurace,
2. ukládání historie příkazů, které uživatel provedl,
3. doplnění *undo* a *redo* operací,
4. realtime validaci úprav konfigurace.

Z pohledu nových možností aplikace, které jsem implementoval v diplomové práci, je v plánu příprava modulárního rozšíření pro:

1. modul Nemea[45],
2. dokončení rozšíření modulu pro konfiguraci linuxového systému.

Další náměty a požadavky na vylepšení by měly přicházet s větším rozšířením aplikace mezi další uživatele. Díky zveřejnění aplikace na GitHubu jsme již nyní získali zpětnou vazbu od několika uživatelů, kteří si *NetopeerGUI* chtěli vyzkoušet.

[45] `https://www.liberouter.org/nemea/`

Literatura

[1] Enns, R.: NETCONF Configuration Protocol. RFC 4741 (Proposed Standard), Prosinec 2006, [Online; přístup dne 26. 4. 2013]. Dostupné z: http://www.ietf.org/rfc/rfc4741.txt

[2] Alexa, D.: Webové uživatelské rozhraní NETCONF klienta s využitím modelů YANG. 2013.

[3] Bjorklund, M.: YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020 (Proposed Standard), Říjen 2010, [Online; přístup dne 30. 10. 2012]. Dostupné z: http://www.ietf.org/rfc/rfc6020.txt

[4] Enns, R.; Bjorklund, M.; Schoenwaelder, J.; aj.: Network Configuration Protocol (NETCONF). RFC 6241 (Proposed Standard), Červen 2011, [Online; přístup dne 30. 10. 2012]. Dostupné z: http://www.ietf.org/rfc/rfc6241.txt

[5] Netconf Central NETCONF Documentation. [Online; přístup dne 13. 2. 2015]. Dostupné z: http://www.netconfcentral.org/netconf_docs

[6] Lhotka, L.: Konfigurace síťových zařízení pomocí protokolu NETCONF. Duben 2012, [Online; přístup dne 30. 10. 2012]. Dostupné z: http://www.root.cz/clanky/konfigurace-sitovych-zarizeni-pomoci-protokolu-netconf/

[7] Scott, M.; Bjorklund, M.: YANG Module for NETCONF Monitoring. RFC 6022 (Proposed Standard), Říjen 2010, [Online; přístup dne 27. 2. 2013]. Dostupné z: http://www.ietf.org/rfc/rfc6022.txt

[8] The NETCONF Wiki. [Online; přístup dne 13. 2. 2015]. Dostupné z: http://trac.tools.ietf.org/wg/netconf/trac/wiki

[9] YANG-Based Unified Modular Automation. 2014, [Online; přístup dne 13. 2. 2015]. Dostupné z: https://www.yumaworks.com/yumapro-sdk/yumapro-sdk/

[10] Yangcli-pro - YumaWorks. 2014, [Online; přístup dne 13. 2. 2015]. Dostupné z: https://www.yumaworks.com/yangcli-pro/

[11] Cesnet TMC group: Netopeer. [Online; přístup dne 11. 3. 2013]. Dostupné z: https://www.liberouter.org/technologies/netconf/

[12] MG-SOFT NETCONF Browser Professional Edition. 2015, [Online; přístup dne 13. 2. 2015]. Dostupné z: http://www.mg-soft.si/mgNetConfBrowser.html

[13] JavaScript Web APIs. [Online; přístup dne 24. 2. 2013]. Dostupné z: http://www.w3.org/standards/webdesign/script.html

[14] Garrett, J. J.: Ajax: A New Approach to Web Applications. Únor 2005, [Online; přístup dne 11. 3. 2013]. Dostupné z: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications

[15] Cascading Style Sheets. Únor 2013, [Online; přístup dne 24. 2. 2013]. Dostupné z: http://www.w3.org/Style/CSS/

[16] Websokets. Únor 2015, [Online; přístup dne 27. 3. 2015]. Dostupné z: https://developer.mozilla.org/en-US/docs/WebSockets

[17] MANIPULATING HISTORY FOR FUN & PROFIT. [Online; přístup dne 27. 3. 2015]. Dostupné z: http://diveintohtml5.info/history.html

[18] Toaster. [Online; přístup dne 21. 4. 2015]. Dostupné z: http://www.netconfcentral.org/modulereport/toaster#make-toast.129

[19] How to Setup before and after Filters. [Online; přístup dne 25. 3. 2015]. Dostupné z: http://symfony.com/doc/current/cookbook/event_dispatcher/before_after_filters.html

[20] Cesnet TMC Group: libnetconf - the NETCONF library in C. 2012, [Online; přístup dne 23. 1. 2013]. Dostupné z: https://code.google.com/p/libnetconf/

[21] Fowler, M.: Inversion of Control Containers and the Dependency Injection pattern. Leden 2004, [Online; přístup dne 27. 2. 2013]. Dostupné z: http://martinfowler.com/articles/injection.html

[22] Purchart, V.: Seriál: Jak na Dependency Injection. Červen 2011, [Online; přístup dne 27. 2. 2013]. Dostupné z: http://www.zdrojak.cz/serialy/jak-na-dependency-injection/

[23] Bernard, B.: Úvod do architektury MVC - Zdroják. Duben 2009, [Online; přístup dne 29. 1. 2013]. Dostupné z: http://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/

[24] PHP: Namespaces overview - Manual. Leden 2013, [Online; přístup dne 29. 1. 2013]. Dostupné z: http://www.php.net/manual/en/language.namespaces.rationale.php

[25] About SAML. Červenec 2008, [Online; přístup dne 27. 4. 2015]. Dostupné z: http://saml.xml.org/about-saml

[26] Zamrzla, J.: Testování a tvorba testovatelného kódu v PHP. Srpen 2012, [Online; přístup dne 12. 4. 2015]. Dostupné z: http://www.zdrojak.cz/clanky/testovani-a-tvorba-testovatelneho-kodu-v-php/

[27] Team, C. D.: Acceptance Testing. [Online; přístup dne 12. 4. 2015]. Dostupné z: http://codeception.com/docs/04-AcceptanceTests

[28] SeleniumHQ.org: SeleniumHQ Browser Automation. [Online; přístup dne 16. 4. 2013]. Dostupné z: http://docs.seleniumhq.org

[29] Team, C. D.: Functional Tests. [Online; přístup dne 12. 4. 2015]. Dostupné z: http://codeception.com/docs/05-FunctionalTests

Příloha A

Seznam použitých zkratek

API Application Programming Interface

CRUD Create, Read, Update, Delete

CSS Cascading Style Sheets

DI Dependency injection

JS JavaScript

JSON JavaScript Object Notation

HTML HyperText Markup Language

MVC Model-View-Controller

ORM Object-relational mapping

GUI Graphical user interface

PHP PHP: Hypertext Preprocessor

RPC Remote procedure call

SASS Syntactically Awesome Stylesheets

UI User interface

XML Extensible markup language

XPath XML Path Language

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář obsahující obraz virtuálního stroje
- `src`
  - `impl` ..... zdrojové kódy implementace (klon repozitáře)
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS
- `tests` ..... adresář obsahující přílohy k uživatelským testům

Příloha C

Instalační příručka

Kompletní instalační příručka je uvedena v README souboru GIT repozitáře na stránkách `https://github.com/CESNET/Netopeer-GUI`.

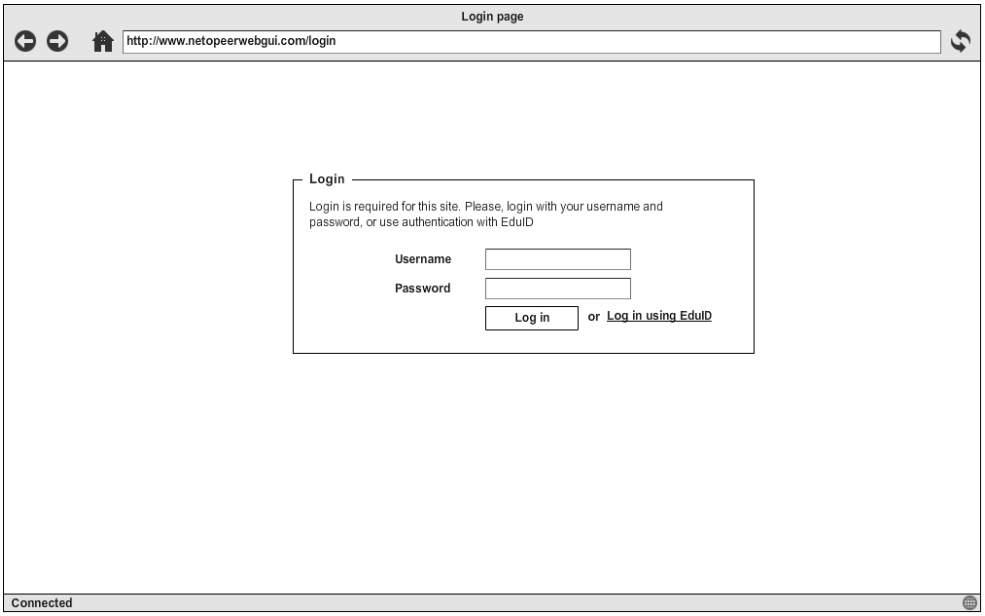
C.1 Seznam závislostí aplikace

- httpd
- *mod_netconf*
- php 5.4
- php-xml
- php-pdo
- php-process
- python
- pyang

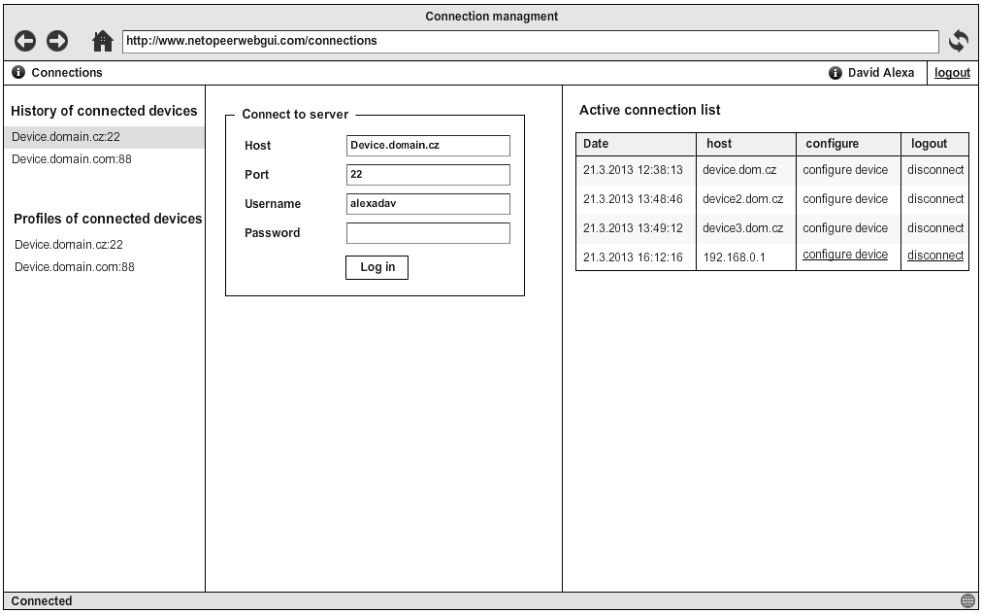
Příloha D

Obrázkové přílohy

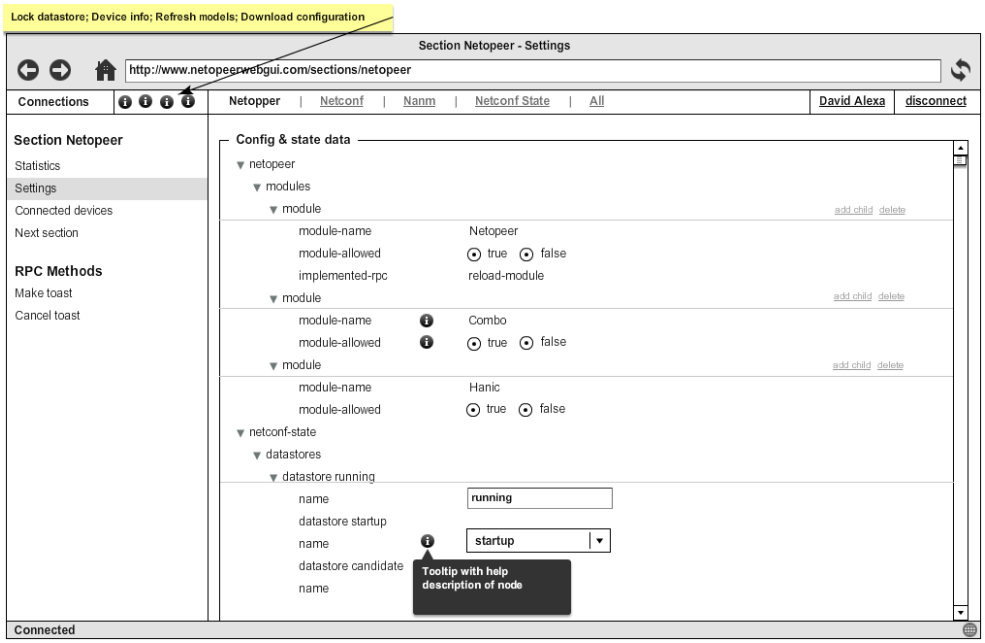
D.1 Wireframy použité jako základ návrhu designu



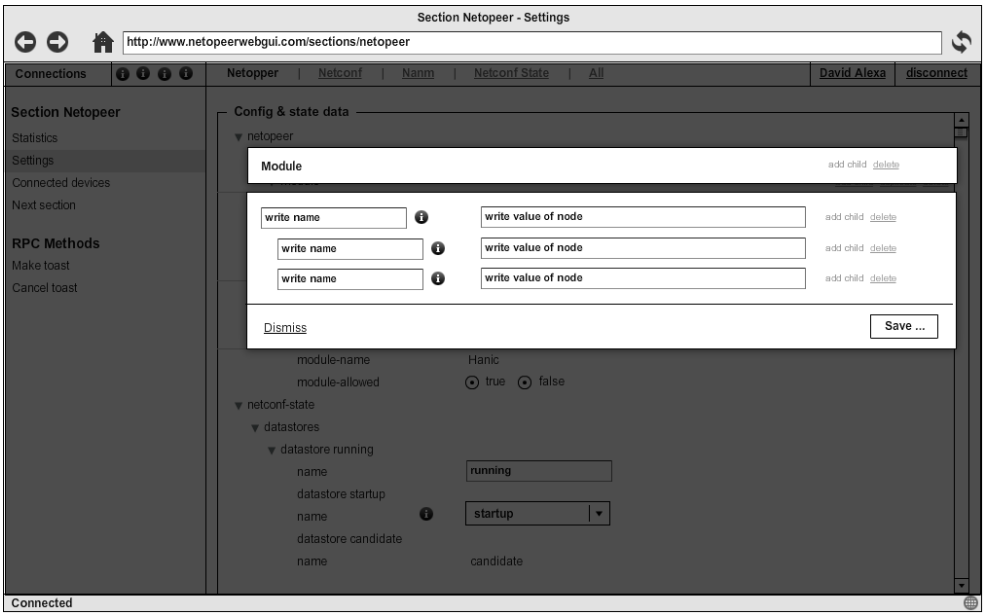
Obrázek D.1: Přihlašovací stránka pro vstup do aplikace. Nepřihlášený uživatel nemá právo zobrazit jakoukoliv jinou stránku.



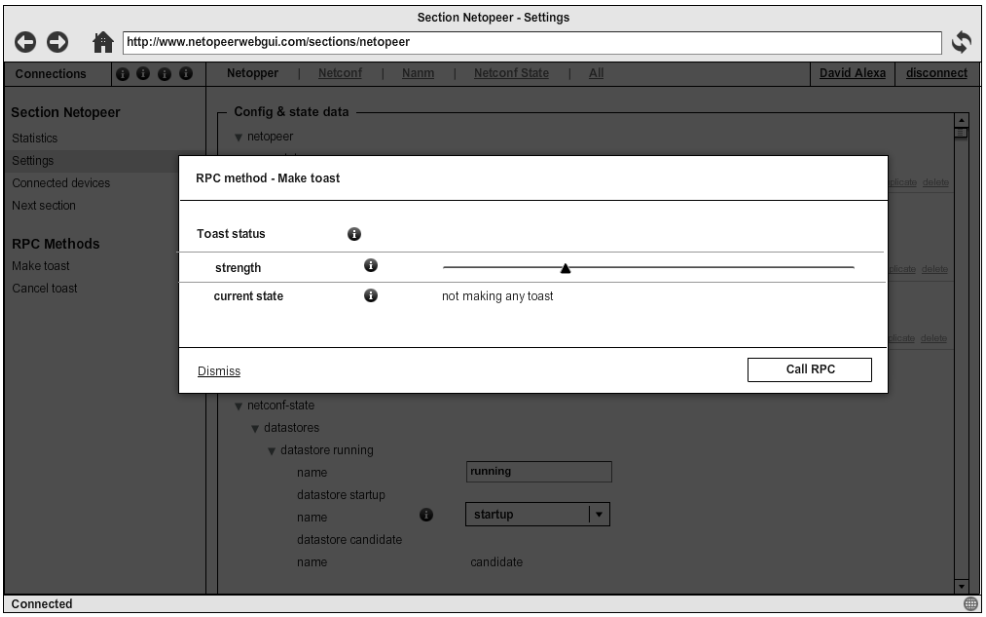
Obrázek D.2: Pokud proběhne přihlášení uživatele úspěšně, zobrazíme rozcestník pro práci se zařízeními. Rozcestník obsahuje historii připojených zařízení a na stálo uložené zařízení (nezávisle na historii). Následuje formulář pro připojení k zařízení a seznam aktuálně připojených zařízení v pravém sloupci. U každého řádku pravé tabulky je možné konfigurovat zařízení nebo se od něj odpojit.



Obrázek D.3: Zobrazení konfigurace zařízení. Horní menu bylo doplněno o uživatelské menu. Levý sloupec obsahuje seznam sekcí a RPC metod zvoleného modulu. V obsahu je zobrazen XML strom s odpovědí *<get>*, spojený s odpovědí *<get-config>* a obohacený sémantickými informacemi z datového modelu.



Obrázek D.4: Modální okno pro přidávání části podstromu obsahuje formulářové prvky pro doplnění názvu a hodnoty uzlu, ale také ovládací prvky pro vložení uzlů nižší úrovně.



Obrázek D.5: Formulář pro volání RPC metod je podobný formuláři pro přidávání části podstromu. Nenabízí však ovládací prvky pro vytváření nových uzlů, pouze předvyplněný strom vstupních parametrů RPC metody.

Příloha E

Testování

E.1 Testovací scénáře

Testovací scénáře jsou uloženy na přiloženém CD.

E.2 Dotazník před testem

1. Umíte anglicky?
   - ano
   - ne
2. Používáte nějaké webové GUI?
   - Ano
   - Ne
   - Na webu se nepohybuji
3. Víte, co je to značkovací jazyk XML?
   - Ano
   - Nikdy jsem o něm neslyšel
4. Umíte jej zobrazit a upravit?
   - Ano, umím jej zobrazit i upravit
   - Umím jej pouze zobrazit, ale nerozumím tomu
   - Neznám jazyk XML
   - Jiné
5. Nastavovali jste někdy domácí router?

- Ano
- Ne

6. Znáte protokol NETCONF?

   - Ano
   - Ne
   - Slyšel jsem o něm, ale nepracoval jsem s ním

Odpovědi u dotazníku před testem byly u všech angličtina ano, GUI ano, XML ano, NETCONF slyšel jsem 4x (plus ano 1x).

E.3 Dotazník po testu

1. Připadá Vám GUI intuitivní?

   - Ano
   - Ne
   - Jiné

2. Vyhovuje Vám rozložení ovládacích prvků?

   - Ano
   - Jiné

3. Měli jste jakýkoliv problém s ovládáním aplikace?

   - Ne
   - Jiné

4. Objevili jste nějaké nedostatky? Něco, co byste zlepšili?

   - Ne
   - Jiné

Výsledky

- intuitivnost - ano
- nějaký závažný problém - ne 4x, ano 1x (způsobeno neznalostí NETCONF, uživatel si vypnul síťové interfaces systému)
- nedostatky - nutná validace a undo

E.4 Výsledky testování pomocí testovacích scénářů

Test 1

Pokud už byl uživatel dříve přihlášen, poté automaticky odhlášen (nebo kliknul na tlačítko odhlásit se), zobrazí se mu část detailu zařízení místo rozcestníku (identifikace problému: chyba načítání obsahu pomocí AJAX).

Test 2

Při vytváření chybí výběrové hodnoty u elementu type (identifikace problému: chyba v datovém modelu). Hodnota elementu chybí také ve výpisu stromu (opět chyba v modelu). Přidání elementu šlo pouze v datastore *start-up* (nutno odsimulovat znovu).

Test 4

Při úpravě elementu IP u jednoho z interfaců v *candidate* a následném uložení konfigurace se element nezměnil, ale zduplikoval (problém: pravděpodobně chybně vygenerovaný edit-config). Při kopírování *candidate* do *running* zůstane zobrazené „nekonečné točítko“ - pád na nižší vrstvě aplikace.

Test 5

Bez problémů.

Test 6

Bez problémů.

Test 7

Po zapnutí modulu a regeneraci struktury menu se modul nezobrazil.

Test 8

Bez problémů