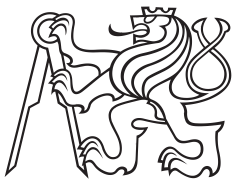
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Automatická správa laboratoře s OS Windows pomocí technologie Puppet

Jan Štěpánek
Softwarové technologie a management

Květen 2016
Vedoucí práce: Ing. Ondřej Votava

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Jan Štěpánek**

Studijní program: Softwarové technologie a management
Obor: Softwarové inženýrství

Název tématu: **Automatická správa laboratoře s OS Windows pomocí technologie Puppet**

Pokyny pro vypracování:

Analyzujte možnosti správy OS Windows pomocí technologie Puppet [1]. Po konzultaci s vedoucím práce zvolte množinu úloh, pro které lze Puppet využít při automatizované správě laboratoře s OS Windows. Tuto množinu implementujte a otestujte její funkčnost.

Seznam odborné literatury:

[1] Arundel, J. (2013). Puppet 3 Cookbook, Packt Publishing
[2] Franceschi, A. (2014). Extending Puppet, Packt Publishing

Vedoucí: Ing. Ondřej Votava

Platnost zadání: do konce letního semestru 2015/2016

L.S.

doc. Ing. Filip Železný, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 26. 3. 2015

Poděkování / Prohlášení

Rád bych poděkoval vedoucímu práce panu Ing. Ondřeji Votavovi za trpělivost a vstřícnost při tvorbě této práce, panu Aleši Kapicovi za jeho rady a konzultace při implementaci práce. V neposlední řadě děkuji mým přátelům a rodině za podporu po celou dobu mého studia.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 10. 5. 2016

.....

Abstrakt / Abstract

Tato práce se zabývá problematikou automatizované správy počítačů s operačním systémem Windows pomocí technologie Puppet. Cílem je představit technologii Puppet, možnosti správy operačního systému Windows a navrhnout pracovní postup, díky kterému lze Puppet udržitelně a systematicky provozovat. Součástí práce je též implementace správy vybraných zdrojů operačního systému Windows v testovacím prostředí.

Klíčová slova: Puppet, správa počítačové infrastruktury, bakalářská závěrečná práce.

This thesis focuses on how to manage computer infrastructure with Windows operating system using Puppet technology. The goal of this work is to introduce Puppet technology, various Puppet's capabilities of Windows management and to propose workflow for infrastructure implementation and further systematic development. This work includes implementation of management of various Windows resources in test environment.

Keywords: Puppet, computer infrastructure management, bachelor thesis.

Title translation: Automatic management of pc laboratory with OS Windows based on Puppet technology

Obsah /

Tabulky / Obrázky

Kapitola 1
Úvod

V této práci představím způsob, jak pomocí Infrastructure as Code (IaC) spravovat počítačovou infrastrukturu. Představím nástroj Puppet jako nástroj IaC pro automatizaci konfigurace a krátce srovnám vybrané alternativní nástroje.

V další části práce podrobně rozeberu možnosti Puppetu pro konfiguraci operačního systému Windows. V prostředí Windows existují nativní nástroje pro správu konfigurace – zejména doména Active Directory a Powershell DSC. Puppet zde představuje zajímavou opensource alternativu.

V samostatné kapitole se budu věnovat problematice automatických testů. Protože Puppet využívá konfiguraci definovanou pomocí zdrojového kódu, lze tento kód velmi pečlivě testovat na různých úrovních.

Nakonec představím konkrétní implementaci konfigurace v testovacím prostředí.

1.1 Osobní motivace

Nejprve se musím přiznat, že jsem duší programátor - nikoliv systémový administrátor. Proto mě téma IaC velice zaujalo jako způsob, jak vnést programátorské návyky do světa konfigurace počítačů. Jako programátoři se snažíme do naší práce zanášet standardy. Učíme se návrhové vzory, učíme se pomocí agilních vývojových metod lépe a efektivněji doručovat software našim zákazníkům, učíme se pomocí automatických testů kontrolovat kvalitu softwaru, učíme se pomocí refaktoru náš software vylepšovat. To vše můžeme pomocí IaC použít i při konfiguraci počítačové infrastruktury.

Kapitola 2
Představení nástroje Puppet

Puppet je nástroj pro správu konfigurace – v anglicky psané literatuře se takové nástroje označují jako *CM Tools* nebo *CCA tools*.

Takových nástrojů je celá řada a v této kapitole bych čtenáře rád seznámil s obecnými vlastnostmi těchto nástrojů a také s odlišnostmi, díky kterým si můžeme vybrat nástroj přesně podle našich požadavků.

2.1 Co to je CM tool

Nástroje pro správu konfigurace lze obecně charakterizovat jako programy, které pomocí konfiguračního souboru provádějí změny konfigurace na vzdálených strojích. Takových programů je celá řada a obecně nelze říci, který je lepší nebo horší. Je třeba vždy uvážit výběr nástroje pro konkrétní použití. V následujících podkapitolách uvedu základní rozdíly mezi zmíněnými programy.

Liší se samozřejmě v jazyku tohoto konfiguračního souboru, dále se mohou lišit v systému distribuce konfigurace na ovládané stroje a také ve způsobu, jak konfiguraci na ovládaných strojích aplikují.

2.1.1 Jazyk konfiguračního souboru

Při výběru konfiguračního nástroje si zároveň vybíráme v jakém jazyce budeme psát vlastní konfiguraci. Každý nástroj vychází z nějakého tradičního programovacího jazyka, namátkou:

- Puppet, Chef [1] - Ruby
- Ansible - Python

Při výběru vhodného nástroje se tedy můžeme řídit naší preferencí daného programovacího prostředí.

2.1.2 Styl programování konfigurace

CM tools nám dávaji na výběr mezi dvěma styly programování.

- deklarativní programování - říkáme *jak* má vypadat stroj po aplikování konfigurace (například Puppet)
- imperativní programování - říkáme *co* má stroj udělat, aby požadované konfigurace dosáhl. (například Chef)

Nutno podotknout, že mezi těmito styly programování není jeden lepší než druhý. Jde o osobní preferenci administrátora, v jakém stylu se mu bude lépe uvažovat [2].

2.1.3 Nastavení klientského stroje

Ovládaný stroj musí být vždy nějak nastaven aby se k němu mohl CM tool připojit a komunikovat s ním. Rozdíly jsou zejména v rozsahu tohoto prvotního nastavení. Puppet vyžaduje na ovládaném stroji nainstalovaný svůj klientský program `puppet`, zatímco například nástroj Ansible si vystačí s prostým ssh připojením a na ovládaném stroji žádný klientský software nepotřebuje [3].

Tyto rozdíly určují zejména počáteční investici do zavedení automatizované infrastruktury – některé nástroje jsou komplikovanější na instalaci než jiné.

2.1.4 Distribuční model pull / push

Máme na výběr ze dvou základních distribučních modelů. V obou existuje centrální prvek, který obsahuje připravenou konfiguraci prostředí.

- Push model - centrální prvek sám aktivně ovládá klientské stroje a přikazuje jim, aby provedly konfiguraci (příklad Ansible ve výchozím nastavení). Zajímavé na tomto modelu je, že musí být dopředu jasné jaké stroje se budou konfigurace účastnit. Jinak by centrální prvek nevěděl, jaké stroje má ovládat.
- Pull model - o konfiguraci žádají klientské stroje a centrální prvek je na vyžádání obsluhuje (příklad Puppet). Narozdíl od pull modelu, zde centrální prvek nepotřebuje dopředu vědět jaké konkrétní stroje bude konfigurovat.

2.1.5 Idempotence

Všechny, zde zmíněné, konfigurační nástroje, včetně Puppetu, aplikují změny konfigurace idempotentně. Tedy opakovanou aplikací konfigurace se stav ovládaného stroje nezmění [4].

Uvažujme například tuto konfiguraci souboru v jazyku Puppet:

```
file { 'c:\temp\file.txt':
    ensure => present,
    content => "generated by puppet manifest"
  }
```

Zdroj `file` neříká, že se bude soubor `file.txt` vytvářet, ale že je třeba zařídit, aby v souborovém systému byl na určené cestě a se zadaným obsahem. To sice může znamenat, že bude soubor vytvořen (pokud chybí), ale také to může znamenat, že se existujícímu souboru změní obsah, nebo se dokonce nestane vůbec nic (pokud soubor popisu již odpovídá). Povšimněme si parametru `ensure` – tím pouze *deklarujeme* požadovaný stav souboru a nemusíme řešit přesný postup jak se tohoto stavu dosáhne.

Na zachování idempotence konfigurace musíme myslet zejména pokud budeme chtít konfigurační nástroj rozšiřovat o naše vlastní funkce nebo pokud budeme potřebovat provést nějaký vlastní příkaz (zdroj Puppetu `exec`). V takovém případě totiž musíme sami zajistit implementaci deklarativního vyjádření konfigurace.

Výše zmíněný příklad v jazyke Puppet je pouze ilustrační – u ostatních CM nástrojůlů je přístup k idempotenci obdobný [5].

2.2 Puppet

Puppet je program pro správu konfigurace počítačových systémů. Byl vytvořen firmou PuppetLabs v roce 2005. Je napsán v jazyce Ruby a pro psaní konfigurace se používá zvláštní DSL jazyk [6], vycházející právě z ruby.

2.2.1 Open Source Puppet vs Puppet Enterprise

Puppet je vydáván ve dvou verzích – Open Source Puppet a Puppet Enterprise [7]. Základní verze Puppetu je dostupná pod opensource licencí [8]. V této verzi je obsažena vlastní implementace jazyka Puppet a všechny nutné nástroje ke zprovoznění Puppet infrastruktury. Veškerý popis jazyka, prostředí i praktické ukázky v této práci se týkají právě varianty Open Source Puppet.

Jako komerční variantu nabízí firma PuppetLabs soubor programů Puppet Enterprise, který nad rámec opensource verze nabízí různé užitečné nástroje na zpříjemnění a zefektivnění použití Puppetu. Enterprise verze nabízí například rozsáhlejší instalační program, nabízí připravené grafické rozhraní na ovládání Puppetu, reporting o běhu Puppetu a samozřejmě profesionální podporu. Kompletní výčet nabízených funkcí placené verze najdeme na oficiálním webu [9].

2.3 Slovníček pojmů z prostředí Puppet

Agent Node

Stroj, zapojený do Puppet infrastruktury, který chceme konfigurovat. Umí konfigurovat sám sebe pomocí `puppet apply` a umí se spojit s masterem v rámci komunikace `agent - master`.

Master

Stroj, zapojený do Puppet infrastruktury, který pro agenty připravuje zkompilovaný katalog.

Manifest

Soubor s kódem Puppetu. Obsahuje vlastní definici konfigurace.

Modul

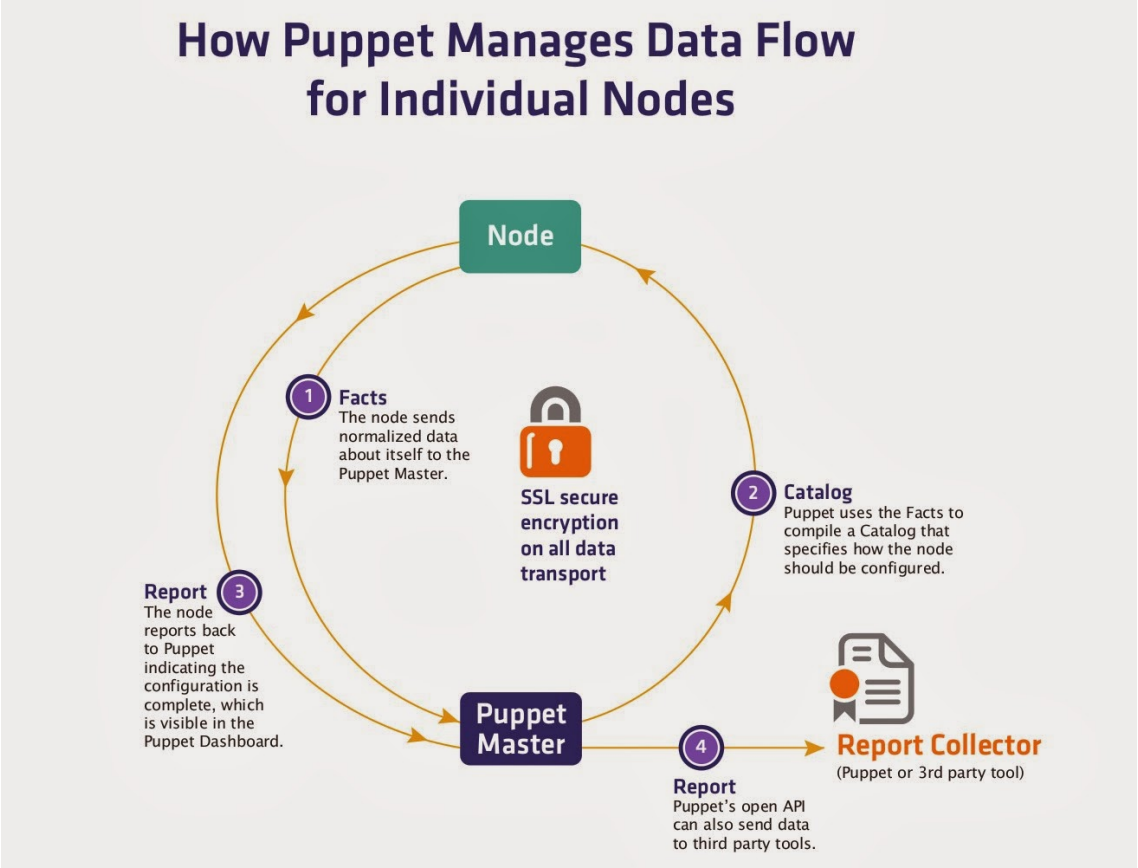
Modul je znovupoužitelná kolekce manifestů. Obsahuje definice typů virtuálních zdrojů Puppetu, soubory a šablony pro běh příslušné konfigurace.

Je to kontejner, který zapouzdřuje nějakou samostatnou, ucelenou konfiguraci. Je to například modul pro konfiguraci Apache serveru [10], nebo konfigurace apt [11] programu v operačním systému Debian.

Moduly si můžeme naprogramovat vlastní nebo můžeme využít již připravených oficiálních i neoficiálních modulů. Oficiální zdroj pro distribuci veřejně dostupných modulů je portál `https://forge.puppet.com/`. Můžeme zde vyhledávat zveřejněné moduly a po přihlášení také vlastní moduly nabízet k použití ostatním. Nejsme ale závislí jen na tomto oficiálním zdroji – můžeme moduly hledat například na `https://github.com/` nebo je udržovat ve vlastním soukromém repositáři. Moduly jsou nakonec jen zdrojové soubory, takže nejsme omezeni žádným proprietárním distribučním kanálem.

Katalog

Výsledná konfigurace, která se aplikuje na ovládaný stroj. Typicky se jedná o kombinaci konfigurací mnoha manifestů. V katalogu jsou umístěny všechny virtuální zdroje z příslušných manifestů, uspořádané do orientovaného, acyklického grafu. Toto uspořádání určuje pořadí aplikace jednotlivých zdrojů.



Obrázek 2.1. Komunikace mezi Puppet Agentem a Masterem. Obrázek převzat z [13]

2.4 Dvě varianty architektury Puppet infrastruktury

Puppet nabízí dva základní způsoby použití. `Puppet apply` a komunikaci `agent - master`.

V případě `puppet apply` se Puppet spouští přímo na konfigurovaném stroji (a žádný jiný stroj není třeba). Nejprve je třeba na stroj nějakým způsobem nahrát soubory s Puppet konfigurací a poté pomocí příkazu

```
puppet apply
```

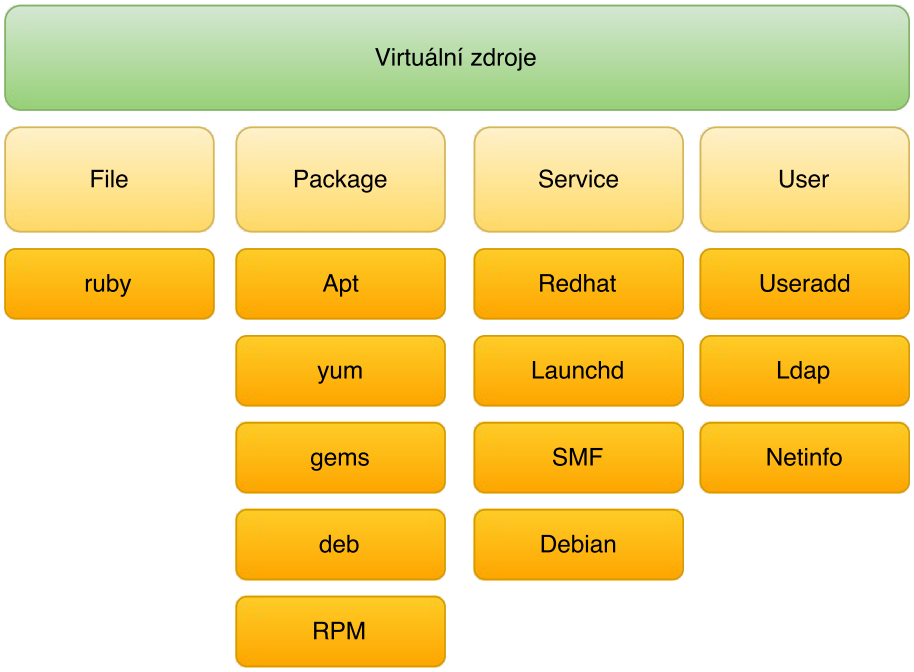
spustíme běh konfigurace [12].

Varianta `agent - master` je určena pro konfiguraci mnoha strojů najednou. Stroje v infrastruktuře jsou rozděleny na dva typy – ovládanému stroji říkáme *Puppet Agent* a ovládajícímu stroji říkáme *Puppet Master*.

Na obrázku 2.1 je znázorněna komunikace mezi agentem a masterem. V bodě 1) zahájí agent komunikaci navázáním spojení s masterem. Předá mu o sobě množství informací, na základě kterých Puppet Master vygeneruje katalog s konfigurací. V bodě 2) posílá master agentovi zpracovaný katalog s konfigurací – agent se ho ihned pokusí spustit. Po aplikaci konfigurace z katalogu agent v bodě 3) posílá masterovi výstup z provedených operací. Tento výstup lze pomocí nástrojů z edice Puppet Enterprise dále zpracovávat a například zobrazovat v grafickém rozhraní pro operátora.

2.5 Zdroje Puppetu jako základní prvky ovládání systému

Zdroje (*resources*) jsou základní entity, se kterými pracujeme při programování v Puppetu. Puppet se nás snaží odstínit od konkrétní implementace reálných zdrojů operačního systému. Používá pro to právě zdroje, které jsou pro různé platformy implementovány *providery*.



Obrázek 2.2. Zdroje puppet.

Puppet nabízí abstrakce například pro práci se soubory, uživateli, systémovými službami, plánovanými úlohami a mnoha dalšími. Kompletní výčet standardních zdrojů nalezneme na oficiálních stránkách [14]. Navíc vybrané zdroje, které se týkají ovládání operačních systémů Windows, podrobněji popíšu ve zvláštní kapitole.

2.6 Podporované platformy

Puppet agent je možné provozovat na nejrůznějších Unix platformách včetně Mac OS, a na moderních verzích operačního systému Windows (verze pro pracovní stanice i servery) [15]. Kompletní podporované verze nalezneme na oficiálních stránkách [16]

Pro provozování Puppet Serveru (Puppet Master) je výběr platforem výrazně menší. Je nutné použít Unix – a to jeden z těchto [17]:

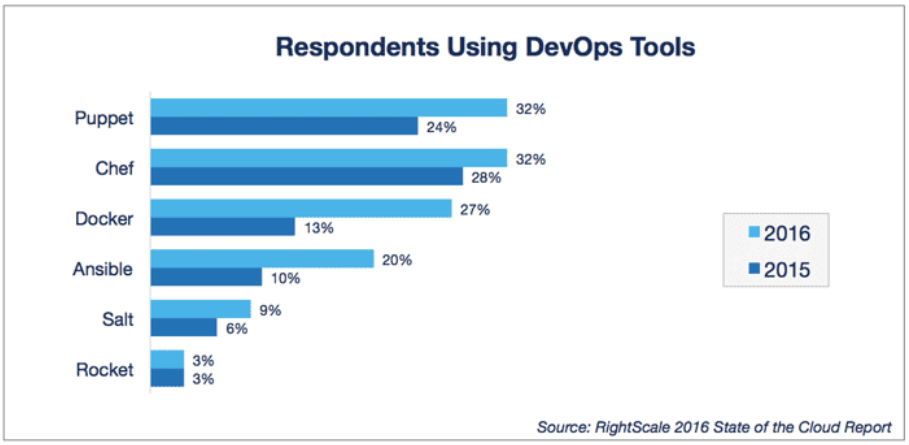
- Red Hat Enterprise Linux
- RHEL-derived distros
- Fedora
- Debian
- Ubuntu

2.7 Jak získat Puppet

Na Unix platformách lze najít oficiální verzi Puppetu i Puppet Serveru v příslušném balíkovacím nástroji. Pro Windows jsou připraveny ke stažení `msi` instalační soubory [18].

2.8 Proč jsem vybral Puppet

Puppet je jedním z nejpoužívanějších [19] nástrojů pro automatizaci konfigurace počítačové infrastruktury.



Obrázek 2.3. Oblíbenost CM nástrojů. Obrázek převzat z [19]

Má velmi propracovaný vlastní dsl jazyk založený na Ruby syntaxi – v kapitole *implementace* předvedu, jak je tento dsl jazyk pohodlný díky jeho *deklarativní* povaze. Programování konfigurace Puppetu používá deklarativní přístup a pro komplikovanější infrastrukturu mi osobně přijde deklarativní model přehlednější. Je to možná tím, že jsem zvyklý na funkcionální programování, které právě deklarativní přístup hojně využívá. Úloha konfigurace se pro deklarativní způsob vývoje, dle mého názoru, velice hodí, protože konfigurace systému je nakonec právě *popis* zamýšleného stavu počítačového systému – nikoliv algoritmus, kterým se ke stavu dostaneme. Samozřejmě pokud bych měl představovat konfigurační nástroje celému vývojovému týmu, ve kterém jsou vývojáři zvyklí výhradně na imperativní programování, Puppet bych patrně nezvolil.

Nejčastější distribuční model Puppetu je *pull* varianta v podobě `agent - master` komunikace. Lze ale použít i variantu *puppet apply*, která obejde centrální prvek Puppet infrastruktury, a docílit tím distribuční varianty *push* – Puppet tedy nabízí to nejlepší z obou modelů.

Protože cílem práce je správa strojů s Windows, je důležitá i velice dobrá podpora Puppetu právě pro tuto skupinu operačních systémů. Jak Puppet umí s Windows pracovat, uvedu ve zvláštní kapitole.

Kapitola 3
Multiplatformní Puppet

Puppet podporuje mnoho operačních systémů z rodiny Unix a také vybrané typy Windows [16]. Pokud to alepsoň trochu jde, snaží se Puppet používat stejné výrazové prostředky na obou platformách. Pro administrátora je tedy výhodné, že mnohé návrhové vzory konfigurace může uplatnit na obou platformách a lépe zúročit čas, který věnoval učení se Puppetu.

Základní stavební kameny systému Puppet jsou nejrůznější zdroje – *resources* [14]. Tyto zdroje jsou implementovány pomocí providerů na konkrétních podporovaných platformách a pro administrátora je tedy připravena příjemná abstrakce nad ovládacími prvky systému. Například zdroj ovládající soubory – *file* – existuje jak pro Unix, tak pro Windows. Liší se právě v implementaci provideru, protože se soubory na Unixu se zachází jinak než na Windows.

Samozřejmě taková abstrakce nezvládne plně odstínit typ operačního systému, ale je příjemné spravovat Unix i Windows podobnými vyjadřovacími prostředky.

V následujících kapitolách bych rád čtenáře seznámil s vybranými zdroji Puppetu a zejména upozornil v čem se liší chování Unixové a Windows verze [20–21].

3.1 Soubory

Zdroj pro ovládání souborů se na obou platformách jmenuje `file`. Společné pro obě platformy je možnost nastavování jména, cesty, obsahu a přístupových práv souboru. Každý z těchto parametrů se ale liší možnostmi, které jednotlivé operační systémy nabízí.

- jméno souboru: narozdíl od Unixu, Windows nerozlišuje mezi velkými a malými písmeny. Puppet toto nijak neupravuje a nechává řešení na administrátorovi.
- lomítka v cestě souboru: Na platformě Windows Puppet automaticky upravuje lomítka na správnou variantu.
- Přístupová práva: Primárně se používá verze oprávnění pro linux – tedy práva v oktalové notaci pro vlastníka, skupinu a ostatní. Windows verze tato oprávnění také dokáže zpracovat a namapuje je na vlastní pojetí oprávnění. Pro podrobnější správu oprávnění k souborům a adresářům však potřebujeme specializované moduly jako je například `puppetlabs/acl` [22] modul.
- Konce řádků: tradičnímu problému při multiplatformní komunikaci se nevyhneme ani v případě Puppetu. Problém je zejména to, že v případě správy Windows spolu vždy komunikuje Puppet Agent a Master, přitom Master nutně musí být Unixový systém. Pro přenos obsahu souborů tedy musíme být velice obezřetní a uchovávat na Puppet Masteru řetězce a šablony souborů se správnými konci řádků. Obecné pravidlo je, že přenos souborů Puppet provádí binárně – tedy zachovává konce řádků takové, jaké jsou uložené u Puppet Mastera. U jiných zdrojů, které mohou upravovat obsah textových souborů (například `hosts` záznamy) se většinou automaticky použijí správné konce řádků daného agenta. Vše je třeba konzultovat s manuálovými stránkami modulů.

3.2 Uživatelé a skupiny uživatelů

Pro ovládání uživatelů a skupin existuje opět společný zdroj Puppetu `user` resp. `group`. Pro Windows je třeba zmínit, že umí ovládat pouze lokální uživatele, nikoliv uživatele Active Directory – k tomu budeme opět potřebovat zvláštní modul.

3.3 Služby

Pro obě platformy existuje zdroj jménem *service* . Liší se v parametrech, pomocí kterých se služby ovládají. Obě platformy mají základní parametr, zda má být služba spuštěna, či zastavena. Unixová varianta umožňuje určit start/stop skripty, zatímco u Windows si volíme *startup type* [23] – parametr udávající zda má být služba spouštěna automaticky, či ručně.

3.4 Záznamy hosts

Ovládání záznamů `hosts` je pro Unix i pro Windows stejné – liší se pouze vnitřní implementace, jak a kam se záznamy zapisují.

3.5 Instalace programů

Správa programů je mezi Unixy a Windows na první pohled naprosto odlišná - kupodivu ale také používá stejný zdroj na obou platformách. Rozdíl pramení zejména z toho, že na Unixu v drtivé většině případů máme nějakého správce instalovaných programů (`apt`, `yum` apod.). Ve standardní instalaci Windows však žádný takový správce není. Musíme tedy sami zajistit, aby měly Windows přístup k instalačnímu *exe* nebo *msi* souboru a poté ho pomocí instalačních parametrů korektně nainstalovat. Distribuci instalačního souboru můžeme zařídit různými způsoby. Můžeme namapovat síťový disk, můžeme soubor stáhnout na disk agenta a po instalaci ho vymazat. Obě tato řešení ale zdaleka nejsou tak pohodlná, jako správci balíků v případě Unixu.

Pro Windows ovšem existuje jistá alternativa k „ručnímu“ stahování programů. Je to aplikace *chocolatey* [24], která pro Windows představuje to co pro Debian `apt` nebo pro Fedoru `yum`. Jedná se o balíčkovací systém pro Windows a Puppet s ním umí díky zvláštnímu provideru pracovat [25]. Nevýhoda ovšem je, že tato aplikace není v čisté instalaci Windows.

3.6 Reboot

Přímo navážeme na předchozí kapitolu o instalaci programů. Na Windows je mnoho programů. které nutně vyžadují po instalaci restart systému – to ve světě Unixu v podstatě neznáme. Nicméně komunita Puppetu se tím nenechala zastrašit a připravila modul pro restart systému. Pomocí obvyklých metod řazení aplikování Puppet zdrojů můžeme například v Puppetu určit, že po instalaci všech programů z manifestu se má stroj restartovat, nebo například k restartu systému má dojít před instalací vybraného programu.

3.7 Spouštění vlastního příkazu

Pro obě platformy se vlastní skripty spouští obdobně zdrojem exec. U Windows existuje zvláštní provider powershell, pomocí kterého můžeme přímo při běhu Puppetu spustit kód v programu `powershell`.

3.8 Plánované úlohy

Zde se verze Windows od Unixů liší více. Unix má na spouštění opakovaných, plánovaných úloh program `cron` – tak se také jmenuje zdroj Puppetu, který tyto úlohy ovládá. Windows ovšem `cron` nemá a tak se jeho zdroj pro spouštění opakovaných úloh odlišuje názvem i formou – `scheduled_task`.

3.9 Windows registry

Jedním z důležitých rozdílů mezi správou Windows a Unixu je forma uchovávání konfigurace v systému. Unix je silně orientovaný na soubory a pro většinu konfigurací používá právě soubory. Z pohledu Puppetu je to velice příjemná věc, protože zdroj na ovládání souborů už má a nemusí proto vytvářet další pro nastavování programů a služeb. U Windows je situace jiná a pro konfiguraci se v drtivé většině případů používají Windows registry.

Puppet pro ovládání registry připravil vlastní modul, pomocí kterého lze přidávat a odebírat klíče z registry a samozřejmě také upravovat typy a hodnoty záznamů.

Kromě samotné konfigurace se Windows registry velice hodí pro získávání informací o systému. Toho Puppet využívá v programu `facter` [26], pomocí kterého se může Puppet rozhodovat v manifestech co přesně na systému vykoná. Pro `facter` si můžeme psát i naše vlastní funkce a v případě Windows jsou právě registry velice vhodná knihovna informací.

3.10 Active Directory

Pro systém Windows existuje tradiční nástroj pro správu entit v síti – Active Directory (AD). Zaměření tohoto nástroje je tedy podobné jako nástroje Puppet. Bylo by ovšem nerozumné považovat Puppet za nesmiřitelnou konkurenci pro AD. Puppet by měl v takovém soupeření velikou nevýhodu, protože je to mnohem mladší systém a musel by přesvědčit správce počítačových sítí, aby opustili zavedené a vyzkoušené řešení AD a dali přednost právě Puppetu. Nezapomeňme, že za AD stojí obrovská mezinárodní korporace a soupeření s ní by tedy jistě nebylo jednoduché [27].

Dle mého názoru je mnohem zajímavější nabídnout stávajícím uživatelům AD nástroj, kterým AD mohou spravovat – nikoliv náhradu, ale užitečný doplněk. A přesně k tomu slouží následující Puppet moduly pro ovládání Active Directory.

3.10.1 Správa Active Directory

Modul pro správu vlastního Active Directory je k dispozici v repositáři `forge.puppet.com` pod názvem `jriviere/Windows_ad` [28]. Tento modul nabízí základní správu AD:

- instalace AD a konfigurace základních parametrů
- správa uživatelských skupin
- správa uživatelů a jejich zařazení do skupin
- správa organizačních jednotek

3.10.2 Modul domain–membership

Modul `trlinkin/domain_membership` [29] slouží k zařazování strojů do domény spravované pomocí AD (*domain membership*).

Pomocí zadaného uživatele s oprávněním přidávat strojedo domény umožní pro stroj určit konkrétní doménu a parametry pro připojení [30].

3.10.3 Modul acl

Pro správu přístupových oprávnění ke zdrojům operačního systému pomocí ACL (*access control list* [31]) slouží modul `puppetlabs/acl` [22].

Příklad použití modulu:

```
acl { 'c:/tempperms':
  permissions => [
   { identity => 'Administrator', rights => ['full'] },
   { identity => 'Users', rights => ['read','execute'] }
 ],
}
```

Modul umožňuje identifikovat uživatele pomocí jejich SID (*securty identifier*) i FQDN (*fully qualified domain name*). Dále podporuje nastavovat práva ke zdroji pomocí množiny ACE (*access control entries*) [32] .

3.10.4 Další moduly pro ovládání prostředí Windows

Modulů, fungujících v prostředí Windows je daleko víc, než je možné v této práci zmínit. I díky aktivní komunitě okolo systému Puppet je na oficiálních zdrojích pro systémy Windows [33] nepřeberné množství užitečných nástrojů[34].

Kapitola 4
Testování

Jako každý jiný zdrojový kód i manifesty Puppetu mohou obsahovat chyby. Můžeme se poučit u programování v tradičních jazycích a do světa správy konfigurace zařadit psaní automatických testů.

Testovat můžeme na více úrovních – můžeme staticky kontrolovat syntax kódu Puppetu pomocí programu `puppet-lint` [35], můžeme psát unit testy pomocí programu `rspec-puppet` a můžeme psát také akceptační testy nad zkonfigurovanými stroji pomocí nástroje `serverspec` [36].

4.1 Statická kotrola kódu

První kontrola kódu začíná linterem - tedy statickou analýzou kódu. Pro Puppet je připraven program `puppet-lint`. Ve standardní konfiguraci obsahuje mnoho užitečných kontrol - od kontroly správného odsazení řádků, přes standardizované pořadí argumentů a vynucování komentářů u definovaných typů, až po odhalení skutečných chyb jako například duplicita v seznamu parametrů modulu.

`Puppet-lint` dokonce umí pomocí parametru `--fix` některé nedostatky sám opravit – například chybný typ odsazení řádků.

Puppet je pro nováčka poměrně záludný jazyk vzhledem ke své deklarativní povaze. Pro programátora zvyklého na procedurální přístup k psaní kódu je `puppet-lint` neocenitelný rádce a pomocník při osvojování „best-practices“.

`Puppet-lint` je otevřený ostaním vývojářům a nabízí na svých stránkách návod, jak si naprogramovat vlastní metody linteru.

4.2 Unit testy

Po statické kontrole kódu jsou na řadě unit testy. Nabízí se použití programu `rspec-puppet` [37]. Tento program je založen na testovacím frameworku pro jazyk Ruby Rspec [38] . `Rspec-puppet` obsahuje veškerou funkcionalitu frameworku rspec a ještě navíc přidává rozšíření pro pohodlné testování Puppet kódu. Při unit testech se žádné nastavování vzdálených strojů neprovádí – běh Puppetu se zastaví při vygenerování katalogu s Puppet zdroji.

Po vygenerování katalogu se kontroluje stav zdrojů, které byly do katalogu zařazeny. Můžeme kontrolovat, zda modul vygeneroval správné typy zdrojů se správnými parametry, můžeme kontrolovat, zda modul skončil výjimkou při zadání nepovolené kombinace parametrů a mnoho dalšího [39].

4.3 Akceptační testy

Akceptační testy provádíme na skutečných, zkonfigurovaných strojích. Pro implementaci testů jsem použil program `serverspec`.

Tento program je, podobně jako `rspec-puppet`, založen na testovacím frameworku Rspec. Obsahuje ale navíc funkce pro testování zdrojů a nastavení systému. Pomocí různých backendů můžeme testy v serverspecu spouštět na různých platformách. Tradičně je podporován Unix, ale můžeme si zvolit i backend pro Docker kontejnery nebo v tomto případě pro stroje s operačním systémem Windows. Pro Unix je kontrola vzdáleného systému implementována ssh příkazy a pro systémy s Windows používá `serverspec` winrm spojení.

Nejzajímavější na programu `serverspec` jsou ovšem dostupné funkce pro testování stavu systému. Kompletní informace nalezneme v manuálových stránkách na internetu. Zde bych ale zmínil několik vybraných typů, které lze použít i na platformě Windows.

- file
- user
- group
- package
- windows_feature
- command
- port
- host
- service
- windows_registry_key

4.4 Test Driven Development

Pomocí testovacích nástrojů, které jsem představil, můžeme i při psaní konfigurace infrastruktury použít moderní techniku psaní kódu – Test Driven Development. Samozřejmě nic nás k psaní testů nenutí (Puppet funguje i bez nich), ale pro bezchybný a udržitelný vývoj jsou testy velice vhodné.

Unit testy a integrační testy oceníme při samotném psaní kódu v Puppetu a akceptační testy psané v nástroji `serverspec` jsou užitečné i při jakékoliv jiné administraci systému – nejsou závislé na použití Puppetu. Pokud bychom se rozhodli vyzkoušet jiný systém pro správu konfigurace, nebo v případě prostředí Windows nativní správu pomocí Active Directory, testy v programu `serverspec` nám poslouží stejně dobře.

Kapitola 5
Testovací prostředí AWS

V předchozích kapitolách jsem srovnával psaní konfigurace s psaním kódu v tradičním programovacím jazyku. V této kapitole představím další podobnost s tradičním programováním – *vývojové prostředí*.

Velmi často programujeme na jiném typu zařízení, než na kterém bude program provozován. Valná většina programátorů programuje na osobním počítači či notebooku, ale přesto výsledek jejich práce funguje na úplně jiných architekturách – programujeme základní desky, roboty, aplikace na mobilních telefonech, mnohatunové stroje ve výrobě – všechny tyto platformy mají společný rys vývoje – použití emulátoru. U konfigurace jsme postaveni před úplně stejný problém. Programujeme kód, který bude fungovat na úplně jiných platformách a proto pro rychlý vývoj potřebujeme opět emulátor cílového systému – *virtualizovaný hardware*.

Od virtualizačního systému potřebujeme zejména rychlé generování strojů pro ladění a testování naší konfigurace. Pro každý use case konfigurace budeme chtít zvláštní izolovaný hardware, abychom se mohli soustředit na konkrétní konfigurační úkol a teprve z vyzkoušených, ověřených postupů skládali funkční celek.

Existuje mnoho služeb a programů, které nám virtualizaci hardware nabízí. Můžeme si zvolit virtualizaci běžící na našem osobním počítači (Virtualbox, Vagrant), nebo zvolit službu, která využívá hw zdroje vzdáleného výpočetního střediska (Amazon Web Services, Microsoft Azure, Google cloud a další). Pro testování Puppetu na Windows jsem se rozhodl použít právě služby vzdáleného datacentra. Díky tomu mám volnost v modelování vícestrojové konfigurace a nejsem vázán dostupným fyzickým hardware mého osobního počítače. Zároveň je možné u vzdálené služby dát přístupové údaje osobám zodpovědným za kontrolu (a nenutit je instalovat si virtualizační program na své počítače).

Puppet je systém pro ovládání zdrojů cílového systému a proto nás jistě nepřekvapí, že i pro virtualizační platformy existuje mnoho Puppet modulů, pomocí nichž lze tyto platformy ovládat. Mezi podporované platformy patří například Amazon Web Services (AWS), Microsoft Azure, VMware, OpenStack nebo Google Compute Engine [40].

Díky programatickému přístupu Puppetu můžeme k vývoji konfigurace infrastruktury přistupovat jako k vývoji běžného programu. Tedy výsledek není konkrétní instalovaná instance stroje, ale kód odevzdaný ve verzovacím systému, který můžeme kdykoliv použít a vyrobit si instance nové.

Puppet nám tak umožňuje uchovávat si přesnou podobu nastavení prostředí, ve kterém naši konfiguraci testujeme a pomocí napojení na api poskytovatele virtualizace můžeme celé prostředí vytvořit a po použití opět odstranit.

5.1 Požadavky na virtualizační platformu

- musí umět spouštět linuxové stroje i stroje s operačním systémem Windows
- musí umět modelovat síťové zapojení strojů, abychom si mohli vyzkoušet `agent - master` komunikaci

- musí být za rozumnou cenu
- musí existovat možnost spravovat prostředí programaticky, nejlépe pomocí Puppet modulů

5.2 Amazon Web services

K testování konfigurace systému Windows pomocí Puppetu jsem si vybral službu Amazon Web Services (AWS). S touto platformou jsem již získal zkušenosti v profesionálním nasazení a je mi proto z uvedených platforem nejbližší. Zároveň splňuje všechny body z výše uvedených požadavků.

AWS je poskytovatel infrastruktury „on demand“. Tedy platí se jen za to, co skutečně používáme. To je velice výhodné v testovacím prostředí, kdy není dlouhodobě spuštěn žádný stroj, ale soustavně se stroje při spuštění testu zakládají a na konci testu se ruší. I při vícestrojové konfiguraci prostředí tak zaplatíme pouze několik minut běhu celého prostředí.

5.3 Služby AWS

V protředí AWS chceme zakládat virtuální stroje a připojovat je do námi zvolené síťové topologie. V následujících kapitolách představím základní stavební prvky prostředí a jejich ovládání pomocí Puppetu.

5.3.1 VPC

Virtual Private Cloud (VPC) je služba pro modelování síťového zapojení virtuálních strojů. VPC v sobě zahrnuje veškeré síťové nastavení prostředí.

- VPC: zapouzdřuje veškerou infrastrukturu v rámci AWS účtu. Určuje AWS region, ve kterém budeme pracovat a udává vnitřní povolený rozsah adres. Příklad založení VPC pomocí Puppet modulu:

```
ec2_vpc { 'puppet-test':
     ensure       => present,
     region       => 'eu-central-1',
     cidr_block   => 10.0.0.0/24
 }
```

- Subnet: představuje podsíť v rámci VPC. Musí existovat alespoň jedna, ale můžeme jich založit víc pro modelování komplikovanější síťové topologie.
- Route table: pro každou podsíť musíme specifikovat routovací tabulku
- Internet Gateway (IGW): prvek pro napojení VPC podsítě do veřejného internetu
- Elastic IP: můžeme alokovat veřejnou ipv4 adresu a přiřadit ji virtuálnímu stroji
- Security Group: můžeme specifikovat jednoduchá pravidla pro povolování a zakazování komunikace po síti – v podstatě je to takový jednoduchý firewall.

5.3.2 EC2

Elastic Compute Cloud (EC2) je služba ovládající virtuální stroje. Zejména je možné pomocí ní vytvářet a mazat virtuální stroje, vytvářet obrazy disků pro zálohování a vytvářet ze strojů šablony ze kterých můžeme zakládat další stroje.

- Instances: vlastní virtuální stroje. Nabízejí se v různých výkonnostních konfiguracích a především si můžeme zvolit různé předinstalované operační systémy. V této práci použiji Debian Jessie a Windows Server 2012.
- Amazon Machine Images (AMI): jsou připravené obrazy disků s nainstalovaným operačním systémem, ze kterých se spouští nové virtuální stroje. Jsou připraveny nejrůznější varianty AMI na AWS Marketplace s předpřipraveným software. Můžeme si ale připravit i naše vlastní šablony a toho využijeme pro základní instalaci Puppetu.
- Volumes: správa pevných disků připojených k instancím.
- Snapshots: vytváření obrazů disků pro zálohu.

5.3.3 Route53

Route53 je dynamický dns server, který umožňuje spravovat záznamy o doméně jak ve vnitřní síti, tak při komunikaci s okolním internetem. Pro ovládání záznamů o doméně je připraveno programatické rozhraní. Toto rozhraní se dá použít například pro implementaci dynamické dns služby, kdy se každý stroj při startu do dns přihlásí a při vypínání odhlásí.

Kapitola 6
Workflow

V této kapitole představím postup vývoje Puppet kódu od programování na lokálním počítači až po nasazení kódu na společná prostředí (testovací nebo dokonce produkční). Puppet přivádí do prostředí Windows správu systému ve formě zdrojového kódu (místo ovládání grafického rozhraní). Stejně jako u jakéhokoliv jiného zdrojového kódu je tedy na místě tento kód verzovat pomocí nějakého verzovacího systému. Pro tento případ jsem zvolil distribuovaný verzovací systém Git.

V předchozích kapitolách jsem se často odkazoval na veřejně dostupné moduly, které můžeme použít pro naši práci. Je načase představit způsob, jak uchovávat závislosti na těchto externích modulech.

6.1 Správa externích modulů

Můžeme si zvolit z více možností jak ke správě modulů přistupovat

- Puppet Module
- Git Submodule
- Puppetfile

6.1.1 Puppet module

První možnost jak se dostat ke sdílenému kódu je pomocí příkazu `puppet module`, který je součátí standardní instalace Puppetu. Pomocí tohoto programu můžeme instalovat moduly zveřejněné na portále `https://forge.puppet.com/` . Nabídka je značně obsáhlá a z veřejných modulů si pravděpodobně vybereme. Může se však stát, že bychom potřebovali nějaký méně známý modul, který najdeme například na `https://github.com`, ale na `https://forge.puppet.com/` zveřejněný není. Stejně tak nám Puppet Module nepomůže v případě, že bychom chtěli sdílet náš vlastní modul mezi projekty a chtěli ho mít pouze ve svém neveřejném repositáři.

6.1.2 Git Submodule

Protože jsem se rozhodl použít git, můžeme zvolit řešení Git Submodule. Valná většina modulů je zveřejněna na `https://github.com` a proto si je můžeme bez problému přidat do našeho projektu jako submoduly. Nevýhoda je ovšem značná náročnost použití submodulů zejména při aktualizaci vzálených repositářů. Navíc je třeba, aby všichni členové týmu se submoduly uměli pracovat.

6.1.3 Puppetfile

Konečně třetí navrhované řešení počítá se zavedením konfiguračního souboru `Puppetfile`. V tomto souboru definujeme externí moduly z repositáře `forge.puppet.com`, můžeme zde uvést i git nebo svn repositáře s příslušnou verzí commitu a nakonec můžeme také specifikovat cestu k lokálnímu modulu. Stažení, respektive aktualizaci

takového Puppetfilu obstará program `r10k`, o kterém ještě bude řeč v následující kapitole. Pomocí příkazu

```
r10k puppetfile install
```

nainstaluje program `r10k` do zadaného adresáře veškeré závislosti z Puppetfilu.

6.2 Udržování konfigurace ve verzovacím systému

Git je velice užitečný nástroj mimojiné díky kvalitní správě různých verzí kódu pomocí větví. Právě větve budou hrát klíčovou roli i v mnou zvoleném postupu.

Když spolupracuje více lidí na jednom puppet projektu, pravděpodobně budou paralelně vyvíjet různé funkcionality. Pomocí gitu si rozdělí svou práci do větví, ale potom přijde problém, jak si na společném prostředí konfiguraci vyzkoušet.

Snadno můžeme vyklonovat git repositář se zdrojovým kódem na Puppet Master do složky `/etc/puppet` a tím zajistit, že Puppet Master aplikuje naši konfiguraci na agenty. Problém nastane v případě, kdy bychom chtěli z jednoho Puppet Mastera distribuovat více verzí konfigurace najednou. Například bychom chtěli vyzkoušet novou verzi modulu ve sdíleném prostředí a nechceme přitom možné chyby zanést na všechny stroje. Hodilo by se, kdybychom si mohli zvolit jeden konkrétní testovací stroj a právě na něm vyzkoušet novou verzi modulu.

Naštěstí autoři Puppetu mysleli i na tento případ a připravili Puppet prostředí (Puppet Environments).

6.3 Puppet Environments

Puppet Master může agentům nabízet více izolovaných konfigurací (prostředí). Agenti si potom dle svého uvážení vyberou jedno prostředí a o to Puppet Mastera požádají. Prostředí se mohou mezi sebou lišit v jedinném parametru, ale i v celém rozsahu kódu.

Povšimněme si prosím, že prostředí je jedním z mála způsobů, jak si agent může zvolit svou konfiguraci a nenechat vše jen na masterovi.

Prostředí mohou být definována staticky v hlavním konfiguračním souboru `/etc/puppet/puppet.conf` [15], pokud ale počítáme s dynamickou množinou prostředí, bude pro nás vhodnější definovat prostředí dynamická. Pro to stačí založit v hlavním adresáři Puppetu adresář `environments` (například `/etc/puppet/environments`) . Agent potom používá prostředí stejně, bez ohledu na to zda jsou definována staticky nebo dynamicky.

Poslední krok spočívá v namapování větví v git repositáři na adresáře dynamických prostředí v Puppet Masteru.

6.4 r10k

Program `r10K` je program napsaný v jazyce Ruby a má za úkol zefektivnit práci a při instalaci Puppet Modulů a prostředí [41]. Je to oficiální program použitý i v komerční variantě Puppetu – Puppet Enterprise.

V předchozí kapitole jsem vysvětlil, jak pomocí Puppetfile spravuje moduly konfigurace a nyní zbývá představit správu dynamických prostředí.

Program r10k dokáže pomocí příkazu `deploy` ze zadaného git repositáře přečíst všechny větve a pro každou z nich vytvořit na určeném místě adresář s obsahem repositáře. Navíc pokud daný repositář obsahuje v kořenovém adresáři Puppetfile, nainstaluje výše zmíněným způsobem všechny potřebné závislosti.

Pokud tedy máme v našem git repositáři větve `production`, `testing`, `dev`, zavoláním příkazu

```
r10k deploy environment -p
```

vytvoříme tuto adresářovou strukturu:

```
/etc/puppet/environments
/etc/puppet/environments/production
/etc/puppet/environments/testing
/etc/puppet/environments/dev
```

Puppet Master potom může nabízet agentům ke konfiguraci tři různá prostředí.

Kapitola 7
Implementace

V této kapitole demonstruji použití systému Puppet při správě počítačové sítě s počítači s operačním systémem Windows. Zaměřím se zejména na pracovní postup vývoje konfigurace a na představení zdrojového kódu konfigurace.

7.1 Specifikace úlohy

Představím modelový příklad, na kterém demonstruji základní techniky a přednosti systému Puppet.

Mějme počítačovou učebnu, ve které probíhá výuka programování. Potřebujeme zajistit, aby na počítačích byly nainstalovány programy pro výuku (Notepad++ a NetBeans), aby se mohli studenti na počítače přihlásit – tedy aby každý měl svého uživatele. Na konci výuky bude pro studenty připraven test – je tedy třeba na stroje rozdistribuovat zadání. Aby ale studenti neopisovali, budou jejich stroje předem rozděleny do dvou skupin a každá skupina dostane zadání odlišné.

7.2 Výběr technologií

Veškeré zdrojové kódy budu udržovat v git repositáři umístěném ve veřejné službě BitBucket `git@bitbucket.org:stepanekj/puppet.git`.

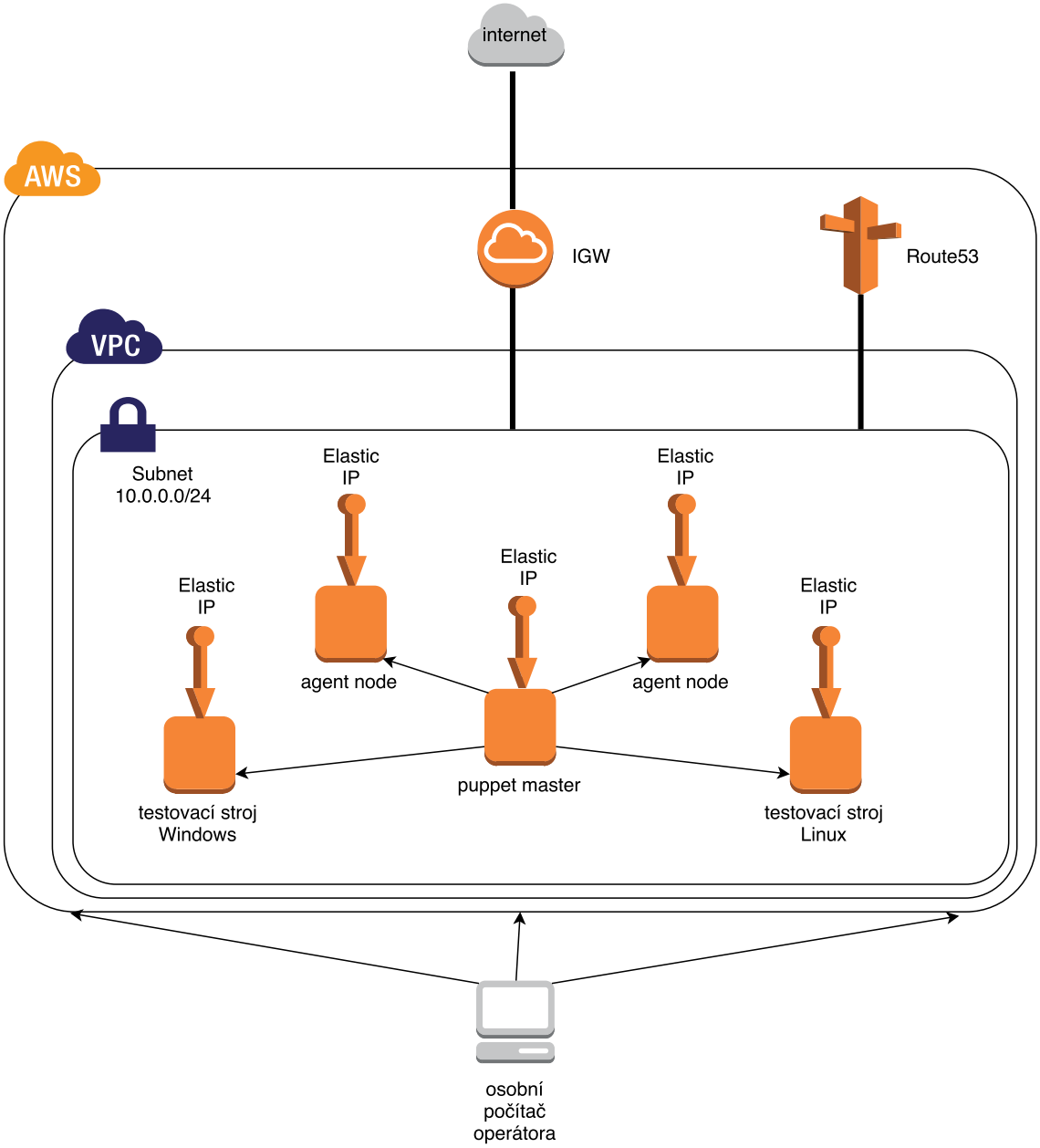
Na základě zhodnocení prostředí AWS z kapitoly Testovací prostředí, jsem se rozhodl použít právě to. Kromě již zmíněných výhod platformy AWS je pozitivní i podpora Puppetu. Puppet nabízí mnoho modulů pro ovládání prvků AWS infrastruktury [42]. Téměř celou konfiguraci prostředí tak mohu udržovat jako Puppet manifest (s výjimkou dynamického registrování strojů do dns Route53).

Jako distribuční model pro tuto úlohu použiji model `agent - master` – tedy bude zapotřebí jeden linuxový stroj pro roli Puppet Mastera a dále spravované stroje s operačním systémem Windows. Pro mastera použiji operační systém Debian ve verzi 8.4 (kódové označení Jessie) a pro agenty použiji operační systém Microsoft Windows Server 2012 R2 Base.

Dále budu samozřejmě potřebovat program `puppet` – zvolil jsem verzi 3.7.2 zejména proto, že je obsažena ve výchozím nastavení Debianu.

7.3 Vývojové prostředí

Na obrázku 7.1 je znázorněna struktura vývojového prostředí. Celé prostředí je izolováno ve vlastním VPC. Uvnitř VPC je definována lokální podsíť, ke které je připojena IGW, aby stroje mohly komunikovat s vnějším internetem. V síti jsou dále umístěny EC2 instance, ke kterým se budu připojovat pomocí ElasticIP. Stroje dostávají při každém spuštění jinou vnitřní IPv4 adresu, ale my musíme zanést propojení strojů do neměnné struktury zdrojového kódu – potřebujeme proto předklad ip adres na stálé dns



Obrázek 7.1. Diagram vývojového prostředí v AWS.

záznamy. To zajistí služba Route53 – dynamicky konfigurovatelný dns server. Bohužel pro dynamické registrování EC2 instancí do Route53 dns serveru zatím Puppet modul není. Při zakládání prostředí jsem tedy zadal příslušné záznamy do Route53 ručně. Dalo by se to jistě zautomatizovat použitím samostatného nástroje, který přihlašuje systém do Route53 při startu a odhlašuje při vypínání.

Dále představím vlastní konfiguraci AWS vývojového prostředí zapsanou jako Puppet Manifest:

```
$vpcName = 'agent-master'
$ensure = 'present'

$debianMasterName = 'debian-master'
```

```
$debianMasterAmi = 'ami-4ebd5221'

aws_vpc { $vpcName:
  ensure  => $ensure,
  vpcName => $vpcName,
}

aws_vpc::ec2_instance { 'debian-master':
  name     => $debianMasterName,
  image_id => $debianMasterAmi,
}


Aws_vpc::Ec2_instance <||> {
  ensure => $ensure,
  subnet => "${vpcName}-subnet",
  security_groups => ["${vpcName}-access"]
}

aws_vpc::ec2_elastic_ip { '52.28.187.150':
  ensure   => $ensure,
  instance => $debianMasterName
}

if $ensure == 'absent' {
  Ec2_instance <||> {
    before => [
      Ec2_vpc_subnet["${vpcName}-subnet"],
      Ec2_securitygroup["${vpcName}-access"]
    ]
  }
}
```

Pro jednoduchost zde uvádím příklad vytvoření jedinného stroje Puppet Master - jak se vytváří více strojů je uvedeno v repositáři práce. Typ `Aws_vpc` je můj vlastní modul, který zapouzdřuje ovládání VPC. Obsahuje definované typy pro vytvoření síťové infrastruktury, pro vytvoření EC2 instance a pro připojování k ElasticIP.

Tento manifest sestaví celé prostředí podle specifikovaných parametrů. Je sestaven tak, aby protředí uměl vytvořit i odstranit pomocí proměnné `ensure`. Je důležité implementovat i odstraňování, abychom mohli po otestování nepoužívané prostředky z aws odstranit (čas po který jsou prostředky spuštěny je zpoplatněn).

Z kódu je patrné, že na sobě jednotlivé zdroje závisí. Pro zapojení EC2 instance musí být například již připravena podsíť. Aplikují se zde *implicitní závislosti*. To je ovšem problém při odstraňování pomocí `ensure => absent` – tehdy je třeba odstraňovat zdroje v opačném pořadí, než při vytváření. Implicitní závislosti totiž neřeší, zda zdroje vytváříme nebo odstraňujeme a aplikuje se vždy.Při odstraňování je ovšem třeba závislosti obrátit. Například nejprve je třeba odtranit EC2 instanci a teprve potom můžeme rušit síť. Proto je na konci manifestu výraz v podmínce `if(ensure => absent)`, který pořadí provádění zdrojů upravuje explicitně.

Dále je třeba zmínit AMI šablony, ze kterých jsou EC2 instance spouštěny. Ty jsem předem připravil tak, že jsem na čistý operační systém nainstaloval program `puppet`, resp. `puppetmaster`. To je první nutný krok, abychom s Puppet infrastrukturou vůbec mohli začít. Dále jsem do AMI obrazu pro Puppet Mastera nainstaloval program `git` a přidal „deploy“ ssh klíč, abych na něj mohl pomocí Gitu nahrát zdrojový kód konfigurace z git repositáře.

Manifest je možné spustit na libovolném počítači, ze kterého se můžeme připojit do našeho AWS účtu pomocí `aws cli` programu. Puppet moduly vyžadují pro přihlášení nastavené proměnné prostředí s přihlašovacími údaji k AWS účtu `AWS_ACCESS_KEY_ID` a `AWS_SECRET_ACCESS_KEY`, které nalezneme ve webovém administračním rozhraní AWS.

7.3.1 Psaní Puppet kódu

Paralelně s vytvářením vývojového prostředí můžeme začít připravovat manifesty i pro hlavní konfiguraci Windows. Vývoj konfigurace probíhá na lokálním počítači pomocí oblíbeného vývojového editoru. Pro mnoho programátorských editorů existují rozšíření pro jazyk Puppet. Při psaní kódu si již na lokálním počítači můžeme postupně ověřovat správnost konfigurace pomocí spouštění `pupet lint` (statická kontrola Puppet kódu) nebo například psaním unit testů v prostředí `rspec-puppet`.

7.3.2 Navázání komunikace agentů s masterem

Pro zahájení komunikace mezi agenty a masterem je třeba provést mezi těmito stroji výměnu a potvrzení ssl certifikátů. Komunikace totiž probíhá zabezpečeně. Každý agent i master si při prvním spuštění vygeneruje dvojici klíčů. Agenti při navázání spojení pošlou k ověření svůj veřejný klíč masterovi a ten si klíč zařadí do skupiny klíčů k ověření. Administrátor potom tyto požadavky na masterovi ověří zkontrolováním otisku klíče a následným příkazem `puppet cert sign <hostname agenta>` potvrdí autenticitu certifikátu. Agentovi se vrátí zpět veřejný klíč mastera – potvrzovat nemusí nic.

Tímto se spustí proces správy konfigurace agenta.

7.3.3 Nasazení kódu na vývojové prostředí

Kód udržovaný v git repositáři potřebujeme nahrát na Puppet Master. Použiji k tomu program `r10k`. Do konfiguračního souboru `/etc/r10k/r10k.yml` na stroji Puppet Master zadáme cestu ke git repositáři a cestu kam chceme kód umístit (například přímo do `/etc/puppet` ,kde ho Puppet očekává). Spuštěním příkazu

```
r10k deploy environment -p
```

na stroji Puppet Master zařídíme nahrání aktuální verze kódu z git repositáře do dané cesty. Program `r10k` zároveň vytvoří správnou adresářovou strukturu v `/etc/puppet/environment`. Pro každou větev v git repositáři připraví vlastní adresář a tím může Puppet pracovat s větvemi gitu jako s Puppet Environments – toho využijeme při nahrávání odlišných verzí zadání testu pro studenty na výukové stroje. V konfiguraci Puppetu každého výukového stroje zadáme jedno ze dvou prostředí a pomocí větví v gitu potom udržujeme rozdíly mezi těmito konfiguracemi.

7.4 Výsledná konfigurace

Úlohu jsem naimplementoval pomocí tří modulů. Ovládají uživatele, instalaci programů a soubory se zadáním.

7.4.1 Modul students

Modul `students` má jeden argument a to seznam studentů. Podle tohoto argumentu založí příslušné lokální uživatele, nejdřív ovšem zařídí, aby existovala potřebná skupina uživatelů `students`. Každému uživateli modul založí vlastní domovský adresář. Bohužel zdroj `user` neumí adresář založit sám a spoléhá se, že již existuje. Proto adresář musíme založit sami, pomocí zdroje `file`.

```
class students(
  $students = []
){
  # local user and local group
 each($students)|$student| {

    file { "C:\\Users\\${student}":
      ensure => directory
    }
    user { $student:
      ensure => present,
      password => '123456Abc',
      groups => ['students', 'Remote Desktop Users'],
      home => "C:\\Users\\${student}"
    }
  }
  group { 'students':
    ensure => present
  }
}
```

7.4.2 Modul classroom_programs

Pro správu programů použiji správce balíků *chocolatey*. Pomocí něj je instalace programů velice snadná – stačí použít zdroj `package`

```
class classroom_programs{
  include 'chocolatey'

  package{ 'notepadplusplus.install':
    ensure => latest,
    provider => chocolatey
  }
  package{ 'netbeans':
    ensure => '7.3',
    provider => chocolatey,
    require => Package['jdk8']
  }
  package{'jdk8':
    ensure => '8.0.92',
    provider => chocolatey
  }
}
```

Všimněme si, že jsem zde musel vyjádřit závislost programu `NetBeans` na `jdk8` – Puppet o této závislosti nic neví a proto zde nenastane *automatický require*.

7.4.3 Modul classroom_test

Nakonec je třeba na vzdálené stroje nahrát zadání úkolu. K tomu nám poslouží modul `classroom\_test`, který díky parametrizované šabloně vyplní zadání pro každého studenta individuálně.

```
class classroom_test(
  $students
){

  each($students)|$student| {

    file { "C:\\Users\\${student}\\test.txt":
      ensure => file,
      require => File["C:\\Users\\${student}"],
      content => template('classroom_test/test.txt.erb')
    }
  }

}
```

Šablona pro zadání s použitým parametrem:

```
Zadání testu A pro <\%=@student\%>
```

Dále mají být stroje rozděleny do dvou skupin tak, aby každá skupina dostala odlišné zadání. To můžeme elegantně zařídit použitím Puppet Environments. V git repositáři k tomu účelu připravíme dvě větve `zadaniA` a `zadaniB`. Na Puppet Masteru potom pomocí příkazu

```
 r10k deploy environment -p
```

vytvoříme následující adresářovou strukturu:

```
/etc/puppet/environments
/etc/puppet/environments/zadaniA
/etc/puppet/environemnts/zadaniB
```

To je přesně struktura, které Puppet Master rozumí a může tak agentům poskytovat dvě různé verze manifestů. V našem případě se manifesty liší práve v šabloně zadání.

Kapitola 8
Otestování funkčnosti implementace

Pro kontrolu správnosti konfigurace připravím sadu automatických testů. Pro každý manifest připravím sadu unit testů. Dále pomocí akceptačních testů, psaných v nástroji `serverspec`, budu testovat strukturu vývojového prostředí a nastavení jednotlivých strojů.

8.1 Unit testy

Pro unit testy používám program `rspec-puppet`. Testy probíhají tak, že se testovaný manifest spustí v odděleném adresáři, kde jsou připraveny všechny závislosti (moduly) pro běh manifestu. Následně se manifest pomocí programu `rspec-puppet` nanečisto spustí (tedy konfigurace doopravdy nemění stav systému), a kontroluje se, zda manifest proběhl jak měl.

Jako příklad unit testu zde uvedu test na třídu `students`

```
describe 'students' do

  let (:title) { 'test_instance' }
  context 'with ensure => present' do
    let(:params) { {
      :students => ['test_student']
    } }

    it { is_expected.to compile }
  end


  let (:title) { 'test_instance' }
  context 'with ensure => ' + instanceState do
    let(:params) { {
      :students => ['test_student', test_student1]
    }
    it {
      should contain_file("C:\Users\test_student").with(
        {
          :ensure => directory
        }
      )
      should contain_file("C:\Users\test_student1").with(
        {
          :ensure => directory
        }
      )
      should contain_user("test_student").with(
        {
```

```
            :ensure => present,
            :password => '123456Abc',
            :groups => ['students', 'Remote Desktop Users'],
            :home => 'C:\Users\test\_student',

          }
        )
        should contain_user("test_student1").with(
          {
            :ensure => present,
            :password => '123456Abc',
            :groups => ['students', 'Remote Desktop Users'],
            :home => 'C:\Users\test_student1',

          }
        )
        should contain_group("students").with(
          {
            :ensure => present,
          }
        )

      }
    end

end
```

8.2 Akceptační testy

Testy jsou ve zvláštní variantě pro každý ovládaný stroj – zde pro stručnost uvedu jeden test pro Puppet Agent.

```
describe user('student1') do
  it { should exist }
  it { should belong_to_group 'students' }
  it { should belong_to_group 'Remote Desktop Users' }
end

describe user('student2') do
  it { should exist }
  it { should belong_to_group 'students' }
  it { should belong_to_group 'Remote Desktop Users' }
end

describe file('c:\Users\student1\test.txt') do
  it { should be_file }
  it { should contain "testu A" }
  it { should contain "student1" }
end
describe file('c:\Users\student2\test.txt') do
  it { should be_file }
  it { should contain "testu A" }
  it { should contain "student2" }
```

```
end

describe user('puppet') do
  it { should exist }
  it { should belong_to_group('Administrators')}
end

describe package('Netbeans IDE 7.3') do
  it { should be_installed }
end

describe package('Notepad++') do
  it { should be_installed }
end
```

8.3 Testování struktury vývojového prostředí

Kromě testování vlastní implementace konfigurace pro Windows ve vývojovém prostředí je vhodné kontrolovat také manifest, kterým prostředí vytváříme. Můžeme ho opět testovat unit testy i akceptačními testy.

Příklad unit testu v `rspec-puppet`, který kontroluje modul `aws_vpc`:

```
describe 'aws_vpc' do
  let (:title) { 'testVpc' }
  context 'with ensure => present' do
    let(:params) { {
      :ensure => 'present',
      :vpcName => 'testVpc'
    } }

    it { is_expected.to compile }
  end

  let (:title) { 'testVpc' }
  context 'with ensure => present' do
    let(:params) { {
      :ensure => 'present',
      :vpcName => 'testVpc'
    } }
    it {
      should contain_ec2_vpc('testVpc')
      should contain_ec2_securitygroup('testVpc-access')
      should contain_ec2_vpc_subnet('testVpc-subnet')
      should contain_ec2_vpc_internet_gateway('testVpc-igw')
      should contain_ec2_vpc_routetable('testVpc-routes')
    }
  end
end
```

V prvním testu kontrolujeme zkompilovatelnost modulu a ve druhém kontrolujeme, že obsahuje definice všech potřebných AWS zdrojů pro vytvoření kompletní síťové infrastruktury.

Akceptační testy jsou vhodné zejména na kontrolu výchozího stavu strojů, které se hlásí pod správu Puppet Mastera. Tyto stroje lze testovat na správné nastavení a spuštění systémové služby `puppet`. Správné nastavení uživatele, pod kterým `puppet` běží a v případě Puppet Mastera můžeme kontrolovat, zda má všechny předpoklady pro stáhnutí zdrojového kódu konfigurace.

Komentovaný příklad testu na kontrolu výchozího stavu systému:

```
# user for connecting to machine
describe user('puppet') do
  it { should exist }
  it { should belong_to_group('Administrators')}
end

# We need git for fetching configuration from git repository
describe package('Git version 2.8.3') do
  it { should be_installed }
end

# We need powershell executing of remote commands
describe Windows_feature('Powershell') do
  it { should be_installed.by("powershell").with_version("4.0") }
end

# ExecutionPolicy for running powershell scripts
describe command('Get-ExecutionPolicy') do
  its(:stdout) { should contain 'Unrestricted'}
end

# We need port 3389 for connecting to machine via winrm protocol
describe port(3389) do
  it { should be_listening }
end

# Obviously we need running Puppet service
describe service('puppet') do
  it { should be_installed }
  it { should be_enabled }
  it { should be_running }
  it { should have_start_mode("Automatic") }
end
```

8.4 Dedikované testovací stroje

Pro běh testů jsem vyhradil zvláštní stroje uvnitř vývojového prostředí. Je třeba na nich udržovat instalace testovacích nástrojů a je tedy vhodné je zapojit do spravované infrastruktury. Další důvod, proč jsem se rozhodl spouštět testy na strojích uvnitř vývojového prostředí je fakt, že zejména akceptační testy v nástroji `serverspec` přistupují k testovaným strojům pomocí jejich hostname a v této práci udržuji dns záznamy strojů pouze ve vnitřní síti.

8.4.1 Rozdělení testů na Windows a Unix

Původně jsem chtěl spouštět všechny testy na jedinném linuxovém stroji. Při tom jsem ovšem narazil na problém při unit testech manifestů pro Windows. Nástroj

`rspec-puppet` se pokusí nanečisto zdroje aplikovat – například při testování instalace programu ho ve skutečnosti neinstaluje, ale zkusí zdroj `package` vytvořit, aby proběhly například kontroly parametrů. K aplikování zdrojů jim ale Puppet musí rozumět a implementace linuxu pochopitelně očekává zdroje ve formátu pro Unix a definici zdrojů Windows vůbec nerozumí. Například pokud se pokusíme validovat vytvoření souboru `c:\soubor.txt` na linuxovém stroji, `rspec-puppet` nám řekne, že je cesta k souboru nesprávně zadaná a že máme zadat plnou cestu od kořene filesystému. Linuxová implementace nerozumí formátu cesty Windows. Podobná situace nastane při volání vlastního příkazu pomocí programu `powershell`. Na linuxu zahlásí `rspec-puppet` chybu neznámého provideru.

8.5 Test driven development

Díky výše zmíněným testovacím technikám můžeme uplatnit i programovací styl *Test Driven Development*. Tedy úzké propojení psaní testů a výkonného kódu, kdy je každá část výkonného kódu napsána jako reakce na neprocházející test. Důsledné testování je velice vhodné právě v případě psaní kódu konfigurace, protože přes všechny automatizující nástroje, znamená rozbití testovacího prostředí výraznější časové zdržení, než v případě rozbití kódu nativního programu, psaného v tradičním programovacím jazyku.

8.6 Výsledky testů

V této kapitole prezenuji výsledky běhu testů zvolené konfigurace systému. Rozdělil jsem testy na tři základní části – testy samotné síťové infrastruktury AWS, testy nastavení strojů pro spouštění testů a konečně testy samotné konfigurace Puppet agentů.

8.6.1 Výsledky testů síťové infrastruktury

use case	výsledek
Modul lze zkompilovat.	PASS
Při zakládání sítě je do katalogu zařazen modul ec2_vpc	PASS
Při zakládání sítě je do katalogu zařazen modul ec2_vpc_subnet	PASS
Při zakládání sítě je do katalogu zařazen modul ec2_vpc_internet_gateway	PASS
Při zakládání sítě je do katalogu zařazen modul ec2_vpc_routetable	PASS
Při zakládání sítě je do katalogu zařazen modul ec2_securitygroup	PASS

Tabulka 8.1. Výsledky unit testů modulu aws_vpc.

V tabulce 8.2 jsou uvedeny testy základních konfigurací testů. Je to právě modul `aws_vpc`, který je za tuto konfiguraci zodpovědný, protože určuje AMI šablony, ze kterých se stroje spouští. Strojů máme celkem pět:

- Stroj na testování Windows
- Stroj na testování Unixu
- Puppet Master
- 2x Puppet agent

Dále kontroluji dostupnost strojů v síti 8.3 . Řádky představují pokusy o spojení stroje v levém sloupci s ostatními. Příznak N/A naznačuje, že komunikace mezi těmito stroji není potřebná a tak se netestovala. Například testovací stroje nepotřebují vidět na žádné okolní stroje.

Dále uvádím seznam use case pro ověření funkčnosti nastavení. Na tento seznam se odkazuji z tabulky 8.2 :

- Usecase1 - stroj se může spojit s vnějším internetem
- Usecase2 - stroj má nainstalovaný program puppet
- Usecase3 - na stroji je spuštěná služba puppet
- Usecase4 - na stroji je nainstalován program git
- Usecase5 - na stroji je nainstalován ssh klíč pro přístup k repositáři

use case	testovací Windows	testovací Unix	Master	Agent1	Agent2
Usecase1	PASS	PASS	PASS	PASS	PASS
Usecase2	PASS	PASS	PASS	PASS	PASS
Usecase3	PASS	PASS	PASS	PASS	PASS
Usecase4	PASS	PASS	PASS	FAIL	FAIL
Usecase5	PASS	PASS	PASS	FAIL	FAIL

Tabulka 8.2. Výsledky akceptačních testů modulu aws_vpc.

zahájení komunikace	testovací Windows	testovací Unix	Master	Agent1	Agent2
Testovací Windows	N/A	N/A	N/A	N/A	N/A
Testovací Unix	N/A	N/A	N/A	N/A	N/A
Master	N/A	N/A	PASS	PASS	PASS
Agent1	N/A	N/A	PASS	N/A	N/A
Agent2	N/A	N/A	PASS	N/A	N/A

Tabulka 8.3. Výsledky akceptačních testů modulu aws_vpc - síťová komunikace.

8.6.2 Výsledky testů testovacích strojů

Pro testovací stroje je manifest velice jednoduchý – pouze nainstalování několika programů. Proto jsem unit testy vypustil a zaměřil se pouze na akceptační testy 8.4.

nainstalovaný program	testovací Windows	testovací Unix
ruby	PASS	PASS
ruby-dev	PASS	PASS
rspec	PASS	PASS
rspec-puppet	PASS	PASS
rake	PASS	PASS

Tabulka 8.4. Výsledky akceptačních testů konfigurace testovacích strojů.

8.6.3 Výsledky testů agentů

Poslední část této kapitoly se věnuje testům spravovaných Puppet agentů.

Bylo třeba otestovat tři moduly - `students`, `classroom_test` a `classroom_programs` .

V tabulce akceptačních testů 8.8 je vidět, jak se liší konfigurace pro stroje `agent1` a `agent2` díky rozdílnému Puppet prostředí.

8.6.4 Ukázka reportovacího nástroje rspec

Testovací nástroj Rspec nabízí různé reportovací nástroje pro zobrazení výsledků běhu testů. Na obrázku 8.1 je výřez html stránky s výsledky běhu testu pomocí html reportovacího nástroje.

use case	výsledek
Modul lze zkompilovat.	PASS
Do katalogu zařazen modul `file` typu adresář	PASS
Do katalogu zařazen modul `file` typu adresář	PASS
Do katalogu zařazen modul `user` se jménem `student1`	PASS
Do katalogu zařazen modul `user` se jménem `student2`	PASS
Do katalogu zařazen modul `group` se jménem `students`	PASS

Tabulka 8.5. Výsledky unit testů modulu students s parametrem `student1` a `student2`.

use case	výsledek
Modul lze zkompilovat.	PASS
Do katalogu zařazen modul file s obsahuje text `Zadání testu A pro student1`	PASS
Do katalogu zařazen modul file s obsahuje text `Zadání testu A pro student2`	PASS

Tabulka 8.6. Výsledky unit testů modulu classroom_test s parametrem `student1` a `student2`.

use case	výsledek
Modul lze zkompilovat.	PASS
Do katalogu zařazen modul `package` s názvem programu `NetBeans`	PASS
Do katalogu zařazen modul `package` s názvem programu `NotepadPlusPlus`	PASS
Do katalogu zařazen modul `package` s názvem programu `jdk8`	PASS

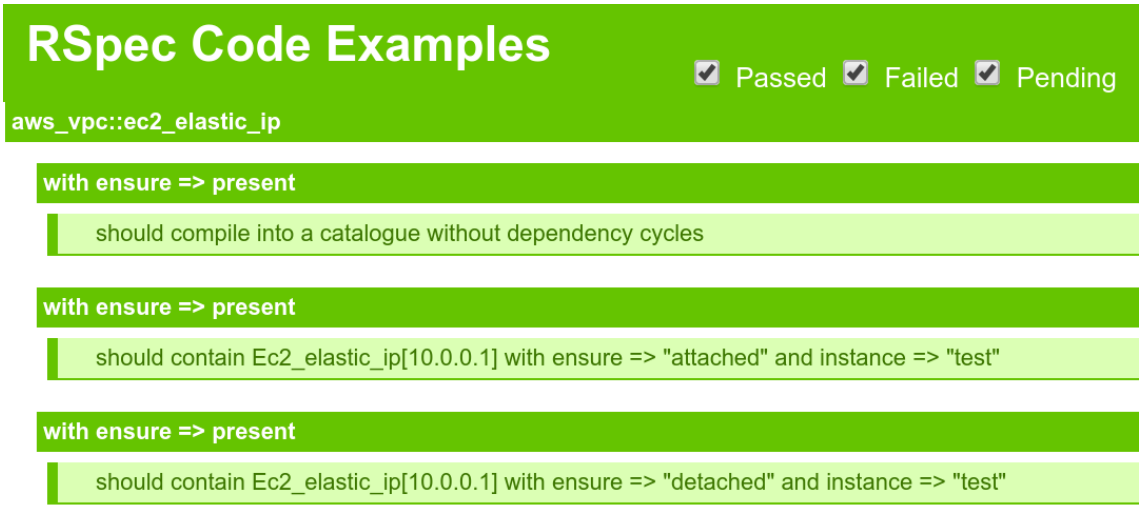
Tabulka 8.7. Výsledky unit testů modulu classroom_programs.

Seznam use case pro akceptační testy s parametrem `student1` a `student2`:

- Usecase1: stroj má založenou skupinu uživatelů `students`
- Usecase2: stroj má založeného uživatele `student1`
- Usecase3: stroj má založeného uživatele `student2`
- Usecase4: na stroji je nainstalován program `Notepad++`
- Usecase5: na stroji je nainstalován program `NetBeans`
- Usecase6: v domovském adresáři uživatele student1 existuje soubor test.txt a obsahuje text Zadání testu A pro student1
- Usecase7: v domovském adresáři uživatele student2 existuje soubor test.txt a obsahuje text Zadání testu A pro student2
- Usecase8: v domovském adresáři uživatele student1 existuje soubor test.txt a obsahuje text Zadání testu B pro student1
- Usecase9: v domovském adresáři uživatele student2 existuje soubor test.txt a obsahuje text Zadání testu B pro student2

use case	Agent1	Agent2
Usecase1	PASS	PASS
Usecase2	PASS	PASS
Usecase3	PASS	PASS
Usecase4	PASS	PASS
Usecase5	PASS	PASS
Usecase6	PASS	FAIL
Usecase7	PASS	FAIL
Usecase8	FAIL	PASS
Usecase9	FAIL	PASS

Tabulka 8.8. Výsledky akceptačních testů Puppet agentů.



Obrázek 8.1. Report běhu `rspec` testu

Kapitola 9
Závěr

V této práci jsem představil problematiku správy počítačové infrastruktury s operačním systémem Windows pomocí nástroje Puppet. Nejprve jsem přestavil nástroj Puppet jako nástroj pro konfiguraci infrastruktury (IaC). Uvedl jsem základní principy Puppetu a srovnal je s vybranými alternativními nástroji.

Dále jsem rozebral konkrétní prostředky, kterými Puppet dokáže spravovat zdroje operačního systému Windows a srovnal použití těchto prostředků na Windows a Unixu.

Prozkoumal jsem prostředí AWS a navrhl, jak ho lze použít jako testovací prostředí pro vývoj konfigurace. Zároveň jsem ukázal, jak lze použít samotný Puppet pro ovládání AWS.

S testovacím prostředím úzce souvisí automatické testy. Představil jsem rozdělení testů na několik kategorií – statická kontrola zdrojového kódu Puppetu, unit testy a nakonec akceptační testy.

Věnoval jsem se udržování kódu Puppetu v git repositáři. Popsal jsem, jak díky konfiguračnímu souboru Puppetfile můžeme udržovat seznam závislostí naší konfigurace na veřejně dostupných modulech. Dále jsem předvedl, jak lze efektivně použít Git pro udržování různých verzí konfigurace pro různá prostředí a jak je lze pomocí programu `r10k` snadno nasazovat.

Nakonec jsem implementoval konkrétní příklady konfigurace v testovacím prostředí AWS a ověřil pomocí automatických testů správnost konfigurace.

Literatura

[1] LOSCHWITZ, Martin. Choosing between the leading open source configuration managers. *ADMIN* [online]. 2014(23) [cit. 2016-05-20]. Dostupné z: https://puppet.com/blog/next-generation-of-puppet-module-testing

[2] *Srovnání deklarativního a imperativního stylu správy konfigurace* [online]. [cit. 2016-02-15]. Dostupné z: https://www.upguard.com/blog/articles/declarative-vs.-imperative-models-for-configuration-management

[3] *Srovnání CM nástrojů Puppet a Ansible* [online]. [cit. 2016-05-20]. Dostupné z: https://ifireball.wordpress.com/2015/01/03/comparison-of-puppet-and-ansible/

[4] *Idempotence - wiki* [online]. [cit. 2016-03-10]. Dostupné z: https://cs.wikipedia.org/wiki/Idempotence

[5] TURK, TYLER. *TESTING ANSIBLE IDEMPOTENCY* [online]. 2016 [cit. 2016-04-05]. Dostupné z: http://tylerturk.com/testing-ansible-idempotency/

[6] KANIES, Luke. [online]. 2012 [cit. 2016-05-20]. Dostupné z: https://puppet.com/blog/why-puppet-has-its-own-configuration-language

[7] *Srovnání Open Source Puppet s Enterprise Puppet* [online]. [cit. 2016-05-20]. Dostupné z: https://www.upguard.com/articles/open-source-puppet-vs.-puppet-enterprise-which-is-right-for-you

[8] *Licence Open Source Puppet* [online]. [cit. 2016-05-03]. Dostupné z: https://github.com/puppetlabs/puppet/blob/master/LICENSE

[9] *Srovnání komerční a opensource verze Puppetu* [online]. [cit. 2016-05-20]. Dostupné z: https://puppet.com/enterprise-and-open-source

[10] *Manuálové stránky pro Puppet modul Apache* [online]. [cit. 2016-04-23]. Dostupné z: https://forge.puppet.com/puppetlabs/apache

[11] *Manuálové stránky pro Puppet modul apt* [online]. [cit. 2016-04-12]. Dostupné z: https://forge.puppet.com/puppetlabs/apt

[12] ARUNDEL, John. *Puppet 3 Cookbook*. Birmingham: Packt Publishing Ltd., 2013, 14-29.

[13] FERNANDO, Chanaka Udaya Kumara. *How puppet works in your IT infrastructure* [online]. 2013 [cit. 2016-03-21]. Dostupné z: http://soatutorials.blogspot.cz/2013_11_01_archive.html

[14] *Manuálové stránky virtuálních zdrojů Puppet* [online]. [cit. 2016-05-03]. Dostupné z: https://docs.puppet.com/puppet/3.7/reference/type.html

[15] TURNBULL, James a Jeffrey MCCUNE. *Pro Puppet*. Birmingham: Apress, 2013, 7-11, 65-88.

[16] *Diagram komunikace mezi Puppet Masterem a Puppet Agent Nodem* [online]. [cit. 2016-04-05]. Dostupné z:
https://docs.puppet.com/guides/platforms.html

[17] *Puppet supported platforms* [online]. [cit. 2016-05-10]. Dostupné z:
https://docs.puppet.com/puppetserver/2.4/services_master_puppetserver.htmlsupported-platforms

[18] *Nabídka instalačních souborů Puppetu pro Windows* [online]. [cit. 2016-05-03]. Dostupné z:
https://downloads.puppetlabs.com/windows/?_ga=1.109402881.525074775.1462901197

[19] WEINS, Kim. *Cloud Computing Trends: 2016 State of the Cloud Survey* [online]. 2016 [cit. 2016-05-10]. Dostupné z:
http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2016-state-cloud-survey

[20] *Představení podpory Puppetu pro Windows* [online]. [cit. 2016-05-20]. Dostupné z:
https://docs.puppet.com/windows/

[21] BENNETT, Liam. *Puppet on Windows* [online]. 2013 [cit. 2016-05-20]. Dostupné z:
http://www.liamjbennett.me/post/2013-10-06-puppet-on-windows-part-1/

[22] *Modul pri ovládání acl přístupu* [online]. [cit. 2016-04-25]. Dostupné z:
https://forge.puppet.com/puppetlabs/acl

[23] *Oficiální dokumentace pro Windows services* [online]. [cit. 2016-05-20]. Dostupné z:
https://msdn.microsoft.com/en-us/library/system.serviceprocess.servicestartmode(v=vs.110).aspx

[24] *Správce programů pro Windows* [online]. [cit. 2016-05-20]. Dostupné z:
https://chocolatey.org/

[25] REYNOLDS, Rob. *Chocolatey: using Chocolatey with Puppet* [online]. 2016 [cit. 2016-05-20].
Dostupné z:
https://puppet.com/blog/chocolatey-using-chocolatey-puppet

[26] *Program facter pro získávání infromací o systému* [online]. [cit. 2016-04-25]. Dostupné z:
https://github.com/puppetlabs/facter

[27] REYNOLDS, Rob. *Puppet: Making Windows Awesome Since 2011* [online]. 2014 [cit. 2016-05-20]. Dostupné z:
http://codebetter.com/robreynolds/2014/08/06/puppet-making-windows-awesome-since-2011/

[28] *Modul pri ovládání Active Directory* [online]. [cit. 2016-05-20]. Dostupné z:
https://forge.puppet.com/jriviere/windows_ad

[29] *Modul pro ovládání příslušnosti do Active Directory* [online]. [cit. 2016-05-03]. Dostupné z:
https://forge.puppet.com/trlinkin/domain_membership

[30] *Dokumentace parametrů připojení do Active Directory* [online]. [cit. 2016-04-10]. Dostupné z:
https://msdn.microsoft.com/en-us/library/aa392154(v=vs.85).aspx

[31] *Oficiální dokumentace k ACL* [online]. [cit. 2016-05-20]. Dostupné z:
https://msdn.microsoft.com/en-us/library/windows/desktop/aa374872(v=vs.85).aspx

[32] *Představení přístupu Puppetu k ACL* [online]. [cit. 2016-05-20]. Dostupné z:
https://puppet.com/blog/managing-permissions-on-windows-access-control-lists

[33] *Nabídka Puppet modulů pro windows* [online]. [cit. 2016-05-20]. Dostupné z:
https://forge.puppet.com/tags/windows

[34] *Soubor užitečných Puppet modulů pro Windows* [online]. [cit. 2016-05-20]. Dostupné z:
https://github.com/puppetlabs/puppetlabs-windows

[35] *Testovací knihovna pro Puppet* [online]. [cit. 2016-05-20]. Dostupné z:
http://puppet-lint.com/

[36] RHETT, Jo. *Learning puppet 4*. Sebastopol: O'Reilly Media, Inc., 2015, 177-188.

[37] *Testovací knihovna pro Puppet* [online]. [cit. 2016-05-20]. Dostupné z:
http://rspec-puppet.com/

[38] *Testovac9 framework Rspec* [online]. [cit. 2016-05-20]. Dostupné z:
http://www.rspec.info

[39] PURVINE-RILEY, Branan. *The Next Generation of Puppet Module Testing* [online]. 2012 [cit. 2016-05-20]. Dostupné z:
https://puppet.com/blog/next-generation-of-puppet-module-testing

[40] *Podpora cloudových prostředí v Puppetu* [online]. [cit. 2016-05-20]. Dostupné z:
https://puppet.com/solutions/cloud-management

[41] FRANCESCHI, Alessandro. *Extending Puppet*. Packt Publishing, 2014, 194-197.

[42] *Puppet Modul pro správu AWS* [online]. [cit. 2016-05-20]. Dostupné z:
https://github.com/puppetlabs/puppetlabs-aws

Příloha A
Slovníček

ACL	Access control list
AD	Active Directory
AMI	Amazon Machine Images
AWS	Amazon Web Services
CCA tool	Continuous Configuration Automation
CM tool	Configuration Management tool
DSC	Desired State Configuration
EC2	Elastic Compute Cloud
IaC	Infrastructure as Code
IGW	Internet Gateway
VPC	Virtual Private Cloud