Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Automatizace procesů firmy v oblasti správy a údržby serverů

Plzeň, 2011 Michal Bryxí

Abstrakt

Business processes automation in the area of server maintenance and management

Companies engaged in Web technologies are using greater number of servers from year to year. Wide expansion of virtualization technologies, also increases the number of server instances. As time passes, documentation of production environment often becomes obsolete. Manual changes on multiple servers is difficult. Human factor often causes unnecessary failures. Therefore, the need for automation in the field of management and maintenance of servers arise. Software providing this feature should be able to cope with different operating systems. It should be independent of the problem that each distribution solves problems differently. It should automate the widest possible range of daily tasks. This document will show, that implementing such a system in already running production environment involves a much greater number of steps than just implementation of the system. I will also describe some steps leading to the improvement of the production environment as such.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Plzni dne _____ _____

podpis

sfssdfs

Obsah

Seznam obrázků

Listings

Seznam tabulek

Kapitola 1

Základní rozbor

1.1 Cíl práce a seznámení s produktem

Tato práce byla vytvořena s cílem restrukturalizace produkčního prostředí firmy, jež se zabývá vývojem a během softwaru sociálních sítí. Hlavními body zájmu z pohledu serverové infrastruktury jsou: stabilita, výkonnost a failover. Z pohledu správy konfigurací jde primárně o tyto body: robustnost, jednoduchost a rozšiřitelnost. Firma měla v době započetí této práce nastaveny jen minimální procesy pro správu a konfiguraci serverů. Z tohoto důvodu bylo rozhodnuto že primárním prvkem této práce má být *zavedení procesu pro konfiguraci serverů.*[1]

1.1.1 Portál

Hlavním předmětem podnikání firmy, pro níž tuto diplomovou práci vytvářím, je sociální aplikace, na kterou se v této práci budu odkazovat jako na **portál**. Ukázkovou titulní stránku portálu lze vidět na obrázku 1.1[2]. Portál je prodáván jako služba, a z toho vyplývá několik důsledků fungování celé firmy. Společnost musí vlastnit serverovou infrastrukturu nutnou pro běh této aplikace. Firma sama si musí zajistit zabezpečení serverů, jejich zálohování, stabilitu, instalaci aplikace a její aktualizace. Již ze samotné povahy sociálních sítí vyplývá vysoký počet přístupů uživatelů s občasnými špičkami v návštěvnosti portálu. Dále je zřejmé, že odstavení produkce není u uživatelů portálu vítaný jev, a proto je dobré se mu vyhnout. Uživatelé dnes od webových služeb očekávají vysoký standard, a proto je potřeba řešit i otázku rychlosti aplikace. V této práci bude rozebrána výkonnost aplikace z pohledu serverů. Portál nabízí velké množství služeb, namátkou: psaní článků, seskupování do komunit, hromadné rozesílání e-mailů skupinám uživatelů, chat, diskuzní fóra, fotogalerie, videa, statistiky uživatelských aktivit a mnoho dalších. Z široké nabídky

[1] Zjednodušeně řečeno jde o vyvinutí softwaru, jenž by se dokázal postarat o konfiguraci serverů a všech služeb potřebných pro běh aplikace, jež firma vyvíjí.

[2] Portál je upraven do stovek vizuálních variant. Náhled je přiložen pouze pro přiblížení aplikace čtenáři.

Obrázek 1.1: Ukázková titulní obrazovka portálu

služeb vyplývá velká množina odpovědností kdy je potřeba dopředu promyslet důsledky jakékoliv změny na celý systém jako takový. Nikoliv jen na jeho části.

1.2 Dosavadní řešení produkčních serverů

Firma, pro níž je tato práce vytvářena, je již několik let na trhu a dodává své produkty několika stovkám zákazníků. Za tuto dobu bylo již zapotřebí nějakým způsobem řešit otázky kolem infrastruktury serverů a problémů s tímto tématem spojených. Stanovení cílů této práce tedy předcházela dlouhá doba seznamování se s aktuálním produkčním řešením firmy. Pochopení všech procesů firmy je pro takovouto práci velice důležité. Na první pohled nedůležitý detail může zničit i velice dobře a pečlivě připravovaný plán reorganizace. Při implementaci jsme vytvořili velké množství testů a simulací, abychom pochopili jak bude používaný hardware reagovat na extrémní podmínky či jen změnu zátěže. Ačkoliv se na první pohled může jevit vytvoření zátěžových testů jako triviální záležitost, v praxi se opak ukázal být pravdou. Překvapivě velmi častou chybou bylo špatné pochopení cíle testů a interpretace výsledků testů. Takovéto chyby pak způsobovaly nelogické chování systému při implementaci určitého řešení. Získané postřehy a zkušenosti budou rozebrány dále.

Na obrázku 1.2 lze vidět zjednodušené schéma produkce firmy. Servery nepotřebné pro téma této práce jsou z obrázku vynechány. Tento obrázek bude postupně doplňován o další body, jež budou diskutovány v dalších částech práce. Nyní se pokusím přiblížit

účel jednotlivých serverů v naší produkci.

xen02 Vstupní server pro požadavky z internetu. Firewall a Virtual Private Network (VPN) server běží zde. Umístění firewallu do takto „předsunutého“ stroje má výhodu v nezatěžování load-balanceru.

lb01 Load balancer - veškeré požadavky na webové služby jdou přes tento server. Ten rozhodne do jaké části infrastruktury má požadavek směřovat.

web0x Výpočetní servery[3], obsluha HTTP požadavků. Každý tento stroj má připojen NFS svazek z *file01*.

db0x Databázové servery, obsluha SQL dotazů.

dev01 Sestavování a distribuce zdrojových kódů aplikace na výpočetní servery - takzvaný deploy.

process01 Server zpracovávající dlouho běžící úkoly. Hromadné posílání e-mailů, konverze dokumentů, apod...

file01 Sdílený prostor pro data, která musí být dostupná na všech serverech - používá protokol NFS.

ext01 Server pro podpůrné webové stránky, jež přímo nesouvisí s produktem společnosti. Díky viditelnosti zvenku má také vlastní firewall.

mail01 Mail server zajišťující jeden odchozí uzel pro poštu.

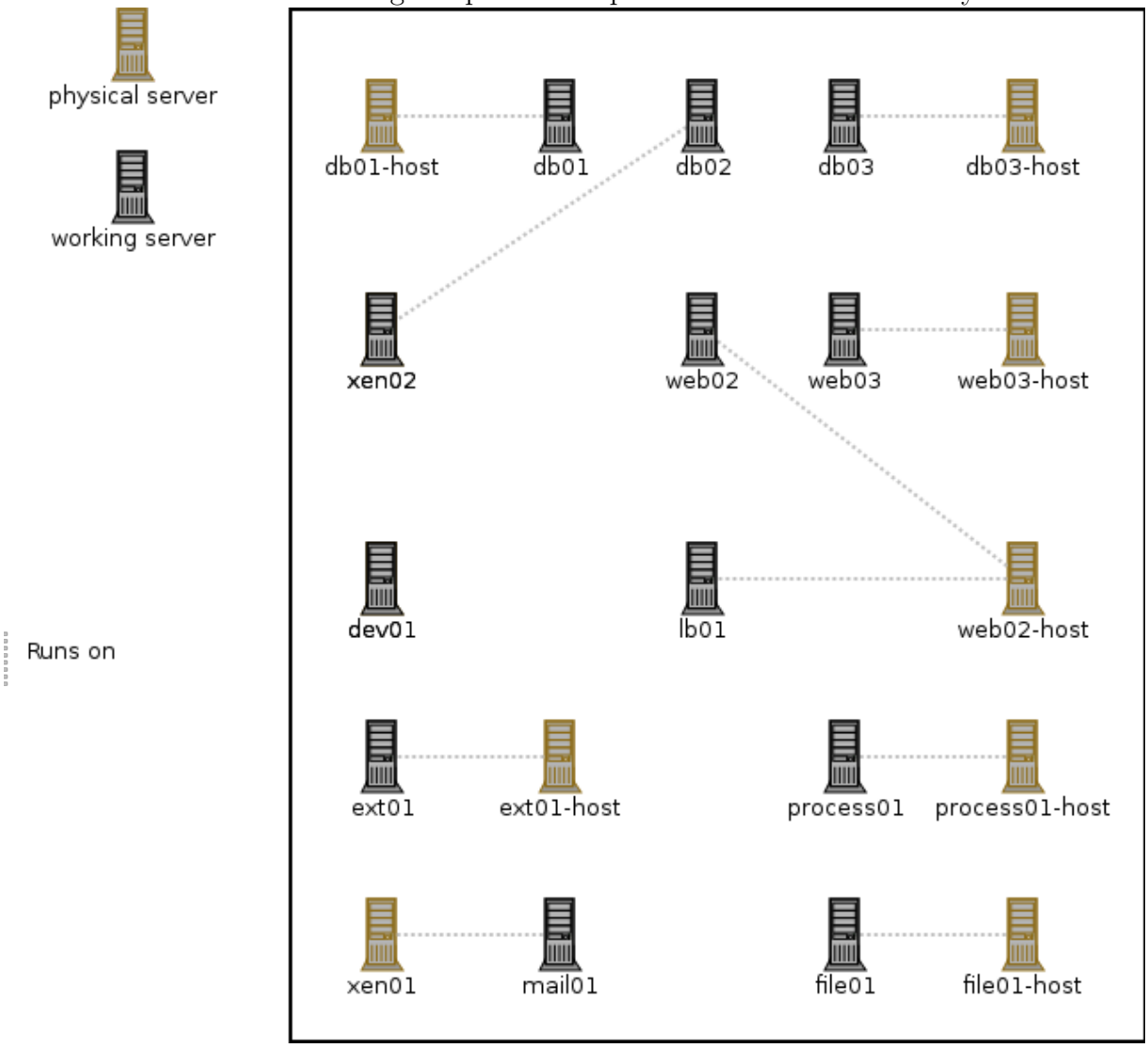
1.3 Výběr vhodného operačního systému

Výběr operačního systému (OS) z velké části definuje výslednou podobu celého produkčního prostředí. To, který z existujících Operační systém (OS) zvolíme, nám vymezuje aplikace, jež bude možné použít. Některé aplikace totiž dokáží běžet jen na určitém typu OS. Naštěstí námi vybraný seznam potřebných aplikací, respektive technologií, není nijak omezující, protože aplikace podporující potřebné technologie existují na všech běžně používaných platformách.

Jedno z prvních omezení, které padne na kteroukoli skupinu rozhodující se pro ten či onen produkt, jsou předchozí zkušenosti. To, jak daná technologie dokázala být nápomocí či překážkou předchozích projektů jasně určuje nasazení této v projektu dalším. Pokud si dokážeme přiznat, že neúspěch není vinou OS, nýbrž naší neznalostí, pak je toto kritérium při rozhodování určitě správné.

[3] Jiné servery v produkci samozřejmě také provádějí „výpočty“, pokud ale bude dále v dokumentu odkazováno na výpočetní server, vždy budou myšleny servery *web0x*.

Obrázek 1.2: Diagram původního produkčního schématu firmy



Náš výběr bude dále zužovat rozpočet projektu. Cena za OS se může pohybovat kdekoliv od nuly až po tisíce korun za instalaci. Do finančních požadavků musíme zanést i náklady na administraci a udržování OS, případné nároky na speciální hardware potřebný pro běh OS, omezení na případný další software, který je nutný pro běh daného OS a další. Všechny tyto hodnoty shrnuje ukazatel *Total cost of ownership* (TCO). Masovější rozšíření tohoto ukazatele se datuje k roku 1987 a od této doby bylo vydáno mnoho studií porovnávajících *TCO* jednotlivých OS. Naneštěstí různé studie dochází k naprosto odlišným závěrům. Například studie Cybersource® - *Linux vs. Windows - Total Cost of Ownership - Comparison*[30] jednoznačně mluví pro řešení postavená na základě linuxových OS. Oproti tomu porovnání společnosti Microsoft® - *Compare Windows to Red Hat*[21] z roku 2003 ukazuje na *TCO* jednoznačně nižší u řešení založených na platformě Windows server. Existují i další studie s opět naprosto odlišnými výsledky.

Vzhledem k osobní zkušenosti, politice firmy a stávajícímu řešení jsme se rozhodli zařadit na seznam přijatelných OS pouze ty, jež jsou založeny na *linuxu*.

Dalším zajímavým rozhodovacím prvkem může být spolehlivost systému. Spolehlivost nám udává, jak dlouho dokáže daný server *bez odstávky* běžet. V roce 2008 provedl *Institute for Advanced Professional Studies* výzkum této hodnoty u nejběžněji používaných OS a vydal zprávu *2008 Server OS Reliability Survey*. Výňatek z této zprávy je vidět na obrázku 1.3. Ve zprávě jsou uvedeny opravdu nízké časy odstávek koncových klientů. Ročně to dělá jednotky hodin. Z předchozích projektů víme, že si včasným nahlášením odstávky služeb a naplánováním odstávky do nočních hodin můžeme dovolit celkovou roční odstávku služeb v mnohem delších časech. Tento údaj nám tedy pouze poslouží jako identifikátor předpokládaných odstávek daných volbou OS.

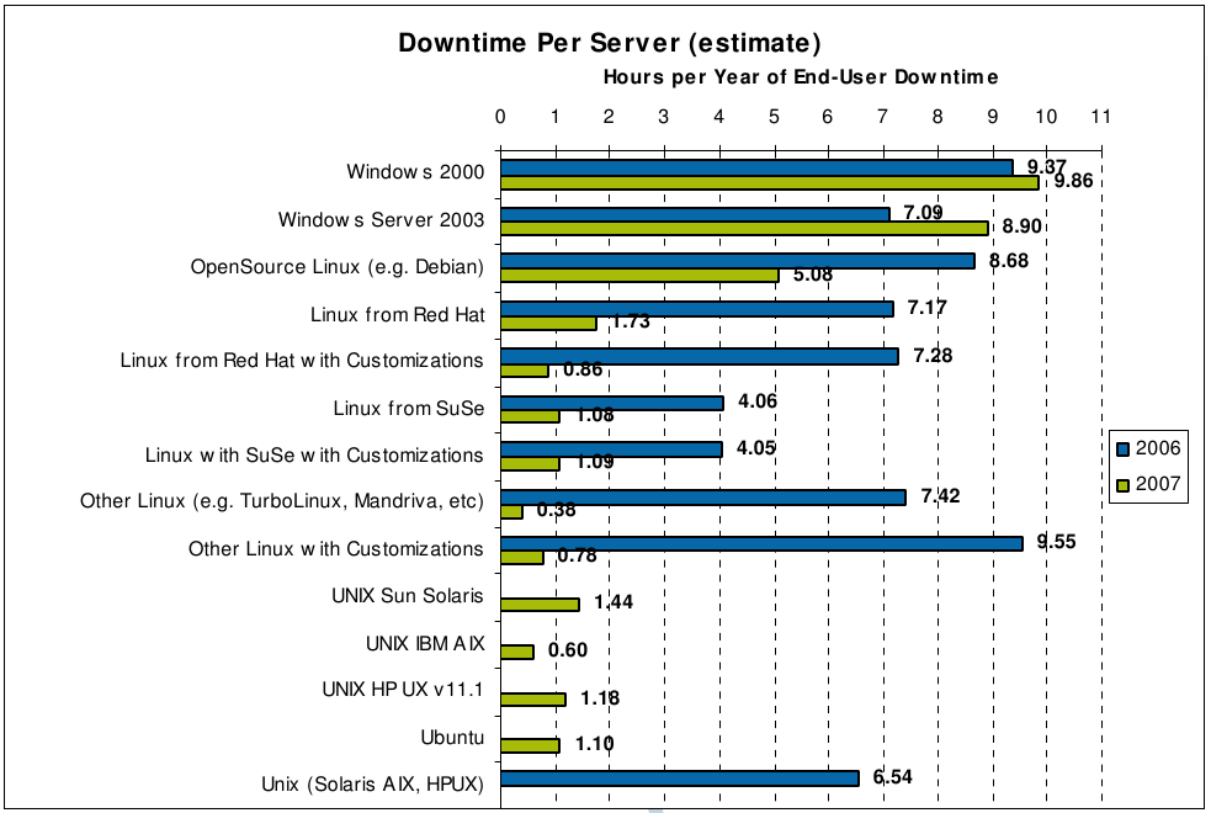
Jelikož v tomto projektu předpokládáme nasazení většího počtu serverů, bude pro nás důležitá schopnost hromadné administrace více strojů, schopnost administrace OS v případě nouze, podpora monitorování a hromadných reportů služeb serveru, a také kvalita a rozsah zdrojů pro popis nestandardních stavů. Naštěstí v dnešní době existuje velké množství takovýchto nástrojů pro všechny běžně používané OS, takže se nemusíme příliš omezovat.

Společně se schopností administrovat serverové instalace nás také bude zajímat možnost aktualizací OS a všech dílčích programů. Tato vlastnost se ukáže jako kritická vždy v nejnevhodnější okamžik, a to ve chvíli, kdy někdo nalezne bezpečnostní chybu v softwaru nainstalovaném na našich serverech. Tehdy se ukáže výhoda pravidelných aktualizací stejně jako nevýhoda zastaralosti softwaru pro některé druhy OS.

1.3.1 Porovnání výhod jednotlivých operačních systémů

Prosím, berte na vědomí, že výhody a nevýhody zde uvedené jsou vázány na **naši situaci** a **naše znalosti**. V jiných situacích se výhody mohou stát nevýhodami a naopak.

Obrázek 1.3: 2008 Server OS Reliability Survey[9]



Windows Server 2008

- Neznámé technologie na poli serverů a virtualizace
- Komerční produkt, zapříčiňuje vysoké přímé počáteční náklady
- Široká možnost podpory
- Rozsáhlá dokumentace na jednom místě - MSDN

Tento OS je pro nás velkou neznámou a nemáme jeho možnosti dobře prozkoumané. Čas potřebný k jeho prozkoumání by tedy byl opravdu vysoký. Jedná se o komerčně prodávaný OS, u kterého předpokládáme nutnost dokoupení velkého množství dalších aplikací, abychom mohli tento systém efektivně provozovat. Tento OS jsme z těchto důvodů vyškrtli z našeho seznamu již na začátku.

Community enterprise operating system - CentOS

- Systém založený na linuxu
- Potřeba stability vynucuje používání staršího softwaru
- Velmi slabá komunitní podpora

- Balíčky na bázi RPM

Tento OS vychází z distribude *Red Hat*. Zkušenost s ním máme z jiných projektů. Bohužel tato zkušenost je naprosto odrazující. Konkrétně se jedná o zastaralost většiny použitého softwaru, chybějící základní balíčky v repozitářích, amatérskou komunitu s nedostatečnou dokumentací, a nepřívětivé chování balíčkovacího systému *yum*. Proto ani tento OS se nám bohužel nejevil jako vhodná volba.

OpenSUSE

- Komunitní distribuce se silným partnerem - firmou Novell
- Rozsáhlá dokumentace na přijatelné úrovni
- Balíčky na bázi RPM

Red Hat Enterprise Linux for Servers

- K jakémukoliv použití zdarma, placená pouze podpora
- Kvůli stabilitě používá starší software
- Neznámá komunitní podpora
- Balíčky založené na RPM

Zde platí většina toho, co bylo napsáno v odstavci pro *CentOS*.

Gentoo

- Velmi kvalitní, aktuální a rozsáhlá dokumentace
- Aktivní, vysoce vzdělaná komunita
- Aktuální balíčky, které občas trpí nedostatečným testováním
- Hardened profil, který zakazuje verze softwaru, u nichž není jistota stability a bezpečnosti
- Balíčkovací systém umožňující u některých balíčků paralelní existenci různých verzí
- Rolling updates
- Náročná distribuce, nevhodná pro začátečníky
- V případě špatné přípravy dlouhá doba pro opravení problému

Gentoo je z mého pohledu velice kvalitní distribucí se špičkovým balíčkovacím systémem. Bohužel daní za jeho flexibilitu je náročnost na znalosti, které musí systémový administrátor mít pro jeho bezproblémovou údržbu. Přes mnoho výhod, které tato distribuce nabízí, jsme se rozhodli od použití této distribuce upustit. Zastupitelnost administrátorů v projektu a jednoduché předání know-how pro nás byly přednější.

Debian

- Projekt s velmi dlouhou tradicí, široce rozšířen
- Rozumný balíčkovací systém s dostatečně aktualizovaným softwarem
- Přijatelná dokumentace, aktivní komunita
- Jednoduchá distribuce
- Balíčky založené na DEB

Díky svému rozšíření a široké uživatelské základně je Debian dobře fungující distribucí. Občas bohužel narazíme na komunitní návody, které jsou buď neaktuální, nebo velice amatérské. Tento jev je daní za velké rozšíření distribuce.

BSD, Solaris, unix

Ostatní zde nejmenované distribuce jsme vyřadili na základě jednoduchého principu naší neznalosti. To, že jsme si vybrali k bližšímu zkoumání některou z výše uvedených distribucí je dáno jistou zkušeností a historickým vývojem. Jiný člověk může mít na věc jiný pohled. Není v silách žádného člověka do detailů prozkoumat všechny OS které existují.

1.3.2 Závěr

Firma, pro niž je tento projekt vytvářen, zpočátku používala *OpenSUSE - 10.3*. Postupně upgradovala potřebné servery až do *OpenSUSE 11.1*. V posledních letech jsou všechny servery postupně migrovány na *Debian Lenny*. Situace rozložení jednotlivých distribucí před započetím prací je vidět na obrázku A.1. Stanoveným cílem tedy je postavit všechny budoucí servery na OS *Debian*. Jako dlouhodobý cíl je pak stanoveno postupně nahradit všechny *OpenSUSE* servery distribucí *Debian*.

1.4 Virtualizace

V posledních několika letech je v IT firmách módou propagovat a implementovat takzvané virtualizované servery. Virtuální server (někdy též nazývaný jako domU, kontejner či hostovaný systém je takový, jenž neběží přímo na hardwaru daného počítače. Přímo

na hardwaru pak běží takzvaný *hardwarový stroj*. Někdy se mu také říká *hostovací OS*, dom0 či Hardware Node (HN). Virtuální server si lze velice zjednodušeně představit jako běžný program, jenž běží v počítači. Tímto programem je celý operační systém. Což způsobuje, že běží jeden operační systém v jiném.

Virtualizace obecně má několik již známých výhod a několik nevýhod, s nimiž je dobré se seznámit před jejím použitím.

1.4.1 Výhody virtualizace

Šetření prostředky Jelikož běží více operačních systémů na jednom serveru zároveň, je velmi často zapotřebí menší množství serverů, než kolik bychom potřebovali bez použití virtualizace. Tato výhoda nabývá na významu, pokud je například potřeba vysokého výkonu pro jeden server ve dne a pro jiný v noci. Výpočetní prostředky se mohou přerozdělit podle aktuální potřeby. Některé servery ze své podstaty nemají vysoké nároky na CPU/RAM. Webhousingové společnosti ale obvykle nenabízejí „malé“ konfigurace. Pokud se více těchto malých virtuálních strojů umístí na jeden silnější HN, pak se může jednat o úsporu v řádu až desítek serverů.

Flexibilita V případě, že chcete přidat výpočetní prostředky do serveru, který není virtualizovaný, obvykle se neobejdete bez fyzické přítomnosti technika v serverovně a vypnutí serveru. V případě virtualizovaného serveru je například přidání více paměti otázkou několika málo příkazů. Obvykle není potřeba vypínat virtuální server.

Jednodušší správa Každý správce serverů se dříve či později setká s problémem, kdy si zruší přístup k serveru a jediná možnost, jak problém opravit, je fyzická přítomnost někoho v serverovně. Samozřejmě slušné server housingové společnosti nabízí možnost vzdálené správy například v podobě zpřístupnění klávesnice a monitoru serveru přes internet. Jenže všechna tato řešení mají určitou prodlevu, než je vzdálená správa instalována. Pokud používáte virtuální servery, pak máte přímý přístup k jejich disku a k „tlačítku“ restart.

Možnost omezení V některých prostředích je výhodná možnost omezit zdroje serveru. V případě fyzického serveru opět jde o komplikovanou proceduru. V případě serveru virtuálního pak o několik příkazů.

Migrace Jelikož i počítače stárnou a jelikož je předpoklad, že Mooreův zákon[4] bude ještě několik let platit, je možnost jednoduchého přesunu OS i se všemi daty velice důležitá.

[4]Složitost součástek se každý rok zdvojnásobí při zachování stejné ceny.

1.4.2 Nevýhody virtualizace

Administrace Virtualizace s sebou přináší spoustu nových vlastností a obvykle i technologií, s kterými musí být administrátor obeznámen. Použití každé další technologie samozřejmě přináší potenciální riziko chyb a problémů.

Výkon Výrobci virtualizačních řešení se snaží uživatele jejich produktů přesvědčit, že *náklady na provoz virtualizace jako takové* jsou nulové. Osobně jsem ze zkušenosti přesvědčen, že toto není pravda, a že v jistých specifických situacích může být overhead virtualizace problém.

Chybovost Jak již bylo uvedeno, použití virtualizace nutně činí celý proces správy serverů složitějším. Administrátor se pak může omylem dopustit chyb, které by u nevirtualizovaného systému řešit nemusel. Druhým typem chyb, které jsou bohužel dle mé zkušenosti relativně časté, jsou chyby v samotné virtualizační technologii. Chyby tohoto charakteru se obvykle *velmi špatně* odhalují a *obvykle* vedou k pádu celého virtualizovaného serveru.

1.4.3 Typy virtualizace

Výběr vhodné virtualizační technologie je velmi důležitý krok při stavbě serverové infrastruktury. Pro bezproblémový běh je vhodné, aby všechny virtualizované stroje používaly stejnou technologii. Například migrace virtuálních strojů mezi stroji fyzickými se tak velmi zjednoduší. Samozřejmě každá z dnes nabízených technologií přináší jisté klady a jisté zápory. Nelze říci, že by v této oblasti existovala technologie, jež by ve všech ohledech předběhla technologie ostatní. V následujícím srovnání uvažujeme pouze open source nástroje, které jsou schopné běžet pod a hostovat *unixové* OS a běží na platformě amd64[5]. Dnes běžně používané technologie v této oblasti jsou: XEN, KVM, OpenVZ.

Velmi pravděpodobně existují i další projekty, které by splňovaly podmínky definované výše. Autor tohoto dokumentu je však buď považoval za nevyhovující, nebo je vůbec neznal.

KVM „V IT je Kernel-based Virtual Machine (KVM) implementací virtuálního stroje využívající jádro operačního systému. Tento přístup obvykle přináší vyšší výkon, než řešení založené na virtuálních strojích, jenž závisí na ovladačích v uživatelském prostoru. KVM se nejčastěji vztahuje k infrastruktuře v Linuxovém kernelu. KVM nabízí nativní virtualizaci na *x86* procesorech, jež poskytují rozšíření *Intel VT-x* či *AMD-V*. Linuxové jádro *2.6.20* jako první obsahovalo implementaci KVM. Výhodou KVM je možnost běhu jakéhokoliv druhu OS nezávisle na OS v HN.“ [25]

[5] jinak také známý jako x86_64

OpenVZ OpenVZ je virtualizace založená na principu kontejnerů [15]. Umožňuje vytvořit velké množství izolovaných kontejnerů na jednom fyzickém serveru, přičemž zajistí izolaci procesů. Každý kontejner se chová jako samostatný server. Může být samostatně restartován, mít vlastní uživatele, paměť, procesy i aplikace. Velkou výhodou *OpenVZ* je široká podpora uživatelských skriptů a využívání dalších technologií. Otázka vytvoření diskového oddílu pro umístění kontejneru je tak vyřešena pouhým přidáním přepínače do příkazu pro vytvoření kontejneru. Další velkou výhodou *OpenVZ* je lehkost s jakou se s danou technologií pracuje. Administrátor HN má automaticky k dispozici administrátorský účet všech virtuálních strojů. Přerozdělování výpočetního výkonu se děje za běhu virtuálních strojů, a to včetně obvykle komplikovaného přerozdělování diskového prostoru.

XEN XEN je jednou z nejstarších a nejznámějších technologií na poli virtualizačních nástrojů. Pro běh XENu je zapotřebí takzvaný *hypervizor*, jenž se stará o zavedení a chod dom0. Jelikož se *XEN* zatím nedostal do oficiální větve linuxového kernelu je nevýhodou této technologie nutnost speciálního jádra. Naštěstí tento problém je díky adopci této technologie většinou hlavních distribucí zanedbatelný. XEN má sám o sobě základní nástroje pro správu virtuálních strojů. Umožňují hlídat aktuální využití systému jednotlivými kontejnery, provádět migrace i automaticky nainstalovat OS na virtuální server.

Výběr virtualizační technologie by neměl být ponechán náhodě. Snadno se může stát, že až po implementaci zjistíme že bychom potřebovali vlastnost, kterou námi adoptovaná technologie nenabízí. V případě tohoto projektu ale musel výběr **nejvhodnějšího** ustoupit daleko důležitějšímu kritériu, a to **zachování homogenity prostředí**. Homogenita je v takovýchto projektech velmi důležitá. Administrace více druhů je vždy daleko složitější a přináší větší množství chyb. Jelikož firma má zaběhnuto nemalé množství strojů využívajících technologii **XEN**, zvolili jsme pro budoucí rozšiřování infrastruktury právě tuto technologii. Nutno podotknout, že *XEN* se v této firmě osvědčil a tudíž není problém s jeho dalším nasazováním. Na obrázku 1.2 jsou znázorněny dom0 stroje pískovou barvou a domU stroje barvou šedou. Vazba mezi dom0 a domU je pak znázorněna šedou přerušovanou čárou.

Na schématu si lze povšimnout jedné zvláštnosti, a to virtualizace *1:1*, neboli nasazení jednoho virtuálního stroje na jednom stroji fyzickém. Zkušenost z tohoto nasazení je následující:

- Mírně vzrostla komplikovanost administrace. Místo aktualizace a konfigurace jednoho musí systémový administrátor spravovat dva operační systémy.
- Občasně vznikají pády systému způsobené přímo XENem.

- Za dobu existence produkčního prostředí popsaného výše nebylo zapotřebí provádět migraci stroje. K potřebě provést migraci nedošlo z různých důvodů. Zaprvé přechodem na jinou distribuci OS. Pak také díky potřebě růstu do šířky[6] současně s upgradem serveru do výšky[7]. V neposlední řadě díky zastarání daného řešení, jež vedlo k manažerskému rozhodnutí postavit daný server znovu. Ukazuje se, že v některých případech tento postup ušetří mnoho času.

- Dle studie *A Performance Comparison of Hypervisors* „má použití XENu za následek jistou výkonnostní penalizaci“ [22]. Naše firma nikdy nedělala test propadu výkonnosti díky použití virtualizace, ale dle zkušeností z externích referencí předpokládáme, že bude činit cca $5 - 10\%$ v závislosti na typu použité virtualizační technologie.

1.4.4 Závěr

Díky výše uvedeným důvodům došlo k následujícím změnám v přístupu k administraci a správě serverů.

Postupná eliminace virtualizace *1:1* v místech, kde evidentně není potřeba. Například výpočetní servery (web02, web03) a databázové servery (db01, db02, db03) pro nás jsou jasným příkladem špatně užité virtualizace *1:1*. Tyto *virtuální* servery totiž již naplno využívají všech výpočetních prostředků *serverů fyzických*, na kterých běží. Pravděpodobnost, že tento fyzický server by hostoval nějaký další domU, je tedy naprosto minimální. Současně je minimální pravděpodobnost komplikací při přesunu dané instance OS na nový (výkonnější) hardware. Z povahy použitého server-housingu jde totiž pouze o sekvenci úkonů: vypnutí serveru, vyjmutí pevných disků, vložení pevných disků do nového serveru, zapojení nového serveru. Dále je pravděpodobnost nutnosti migrace dané instance OS k jinému poskytovateli server-housingu velmi malá, a to z důvodu nutnosti udržení *minimální odezvy* mezi jednotlivými servery. Odezva na místní *100Mbit* lince je obvykle okolo *0.5ms*, odezva na *1Gbit* lince pak bývá *0.1ms* a doba odezvy mezi jednotlivými server-housingovými společnostmi nabývá až *30ms*. Takovéto zpoždění je pro hladký běh aplikace naprosto nepřípustné a je tedy nutné všechny servery *přímo* zodpovědné za běh produkce udržet co „nejblíže“ u sebe.

1.5 Load balancing

V počátku vývoje většiny webových projektů je pravděpodobné, že se aplikace vejde na jeden server - kódy aplikace, databáze, HTTP server, i uživatelská data. Ve chvíli, kdy aplikace přeroste výkon hardwaru serveru se obvykle zakoupí server větší (výkonnější). Pokud aplikace přeroste i tento server, zakoupí se ještě výkonnější a tak dále. Toto kolečko

[6] Růst do šířky - rozdělení služby mezi více serverů. Obvykle spojeno se změnami v topologii.
[7] Růst do výšky - výměna serveru za výkonější.

bohužel nelze opakovat donekonečna. Výkon jednoho serveru je shora omezený a v jednu chvíli se dostaneme do bodu, kdy již dostupné Hardware (HW) technologie nestačí. Tento typ škálování je obecně známý jako škálování do výšky, neboli "scale up".

Proto se obvykle přistupuje ke škálování do šířky nebo-li "scale out". Při tomto typu škálování jsou přidávány do systému další servery a zátěž je rovnoměrně distribuována mezi tyto servery. Výhodou tohoto řešení je, že má obecně daleko větší škálovatelnost[8]. Navíc navýšení výkonu touto metodou v zaběhnutém systému nevyžaduje žádnou, nebo pouze minimální odstávku.

Stav infrastruktury, jenž byl před započetím prací na tomto projektu, lze vidět na obrázku A.2. Jedná se o postupnou přirozenou evoluci, kdy zvětšující se služby jsou postupně odsouvány na vlastní servery (*process01*, *mail01*). Tento krok je logický a velmi ulehčuje jak přetíženým výpočetním serverům, tak administrátorům, jež mohou tyto služby obsluhovat zvlášť.

HTTP nebo HTTPS požadavek do infrastruktury firmy vchází přes hraniční stroj *xen02*. Jelikož internet není ani zdaleka poklidné místo s uživateli, kteří by měli dobré úmysly, je nutné předsunout před jakékoliv zpracování dat z vnějšku infrastruktury firewall. *Xen02* pak dále předá požadavek na *lb01*, který slouží k distribuci zátěže rovnoměrně mezi výpočetní servery (load balancing). Za povšimnutí stojí fakt, že dále již putuje pouze HTTP požadavek. Krom výhody značně zjednodušené konfigurace je v tomto řešení výhoda správného rozmístění zátěže. *lb01* pouze přebírá HTTPS požadavky, rozbalí je, určí cílový výpočetní server a pošle dál jako HTTP požadavek. Cílový výpočetní server se tak nemusí starat o odstraňování *SSL/TLS* vrstvy. Další výhodou tohoto řešení je failover, který přináší existence více výpočetních serverů. Pokud je například *web02* vyřazen z provozu, pak jeho práci automaticky převezme *web03*.

Výpočetní servery dále požadují data od databázových serverů. Jelikož velikost jedné databáze zatím nepřesáhla velikost jednoho databázového stroje, nebylo zatím zapotřebí load balancing[9] na úrovni databází řešit. Webový server se rozhodne pro správnou databázi na základě konfigurace. Tyto konfigurace se nachází na sdíleném diskovém prostoru jenž je mountován pomocí protokolu NFS. Tento fakt zachycuje obrázek A.3. Zákaznické konfigurace samozřejmě není nezbytně nutné distribuovat pomocí sdíleného filesystému, ale toto řešení značně zjednoduší jejich správu, kdy změnou na jednom místě dojde k okamžité změně na všech místech.

1.5.1 Závěr

Schopnost aplikace škálování do šířky je pro tento projekt velmi důležitá a z obecného pohledu velmi výhodná. Dává nám relativně jednoduchou cestu zvyšování výkonu. Nelze ale očekávat lineární nárůst výkonu společně s počtem zapojených výpočetních serverů.

[8]Schopnost dané technologie přírůstkovým způsobem zvyšovat sledované parametry v případě, že nastane taková potřeba.

[9]Správnější pojem by zde byl „clustering“.

Brzy se totiž objeví další úzká hrdla v podobě propustnosti sítě, schopností databází či sdíleného diskového prostoru. Problémy, na které jsme narazili při rozšiřování infrastruktury budou rozebrány dále.

1.6 Failover

> V IT se pod pojmem failover rozumí schopnost systému automaticky přepnout na redundantní server, systém či počítačovou síť při poruše nebo abnormálním vypnutí dříve běžící aktivní aplikace, serveru, systému či sítě. Failover nastává bez lidského zásahu a obvykle bez varování [23].

Nad systémem, jenž by jako celek beze zbytku splňoval výše uvedenou definici, by jistě zajásal každý systémový administrátor. Bohužel vybudování takovéto infrastruktury stojí velké množství času a úsilí. Úprava již existujícího produkčního systému, jenž nebyl takto od začátku budovaný, je pak ještě mnohem náročnější. Je proto velmi dobré si na začátku budování rozdělit produkci na části, jež mohou potenciálně selhat, a u těchto pak určit, zda je nutný failover či ne. Příkladem, kdy je dobré mít failover, je výpočetní server webových stránek. Nedostupnost webových stránek je pro firmu, vydělávající na webové platformě, velký problém. Naopak celkem zbytečné je zálohovat server provádějící zpracování dávkových operací. Pozdržení odeslání hromadných e-mailů, či vytvoření náhledů u videí, je velmi dobře tolerovatelné. Nyní by bylo dobré si probrat několik případů, ve kterých se běžně využívá failover.

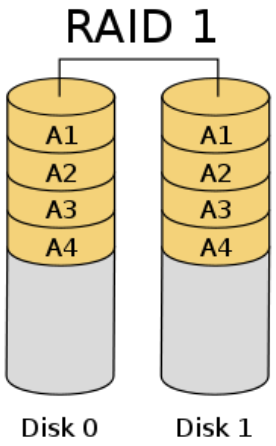
1.6.1 Failover pevného disku

Pokud selže systémový disk serveru, může se doba obnovy havárie pohybovat kdekoliv od několika málo *minut* až po několik *hodin*. V některých případech dlouhá obnova nevadí, ale čas člověka na tento úkon vynaložený je zbytečně ztracený. Nejběžnějším řešením *failoveru pevného disku* je Redundant Array of Independent Disks (RAID). RAID zjednodušeně řečeno vezme data, která by měla být zapsána na disk, a určitým způsobem je zapíše na více disků. Podle způsobu zápisu je pak RAID dělen na podkategorie. Nás bude zajímat *RAID 1*, jenž je dle definice z wikipedie:

> Nejjednodušší, ale poměrně efektivní ochrana dat. Provádí se zrcadlení (mirroring) obsahu disků. Obsah se současně zaznamenává na dva disky. V případě výpadku jednoho disku se pracuje s kopií, která je ihned k dispozici [27].

Ukázku *RAID 1* lze vidět na obrázku 1.4. *RAID 1* nás zajímá především proto, že zajišťuje pro náš případ potřebný *failover*. Současně je díky automatické dodávce serverů se dvěma totožnými disky u námi použitého server-housingu *RAID 1* nejjednodušším a finančně nejméně náročným řešením.

Obrázek 1.4: RAID 1



Ohledně souboje softwarový RAID versus hardwarový RAID toho bylo již napsáno mnoho. Když pomineme články společností vyvíjející moduly hardwarového RAIDu a články vývojářů komerčních linuxových distribucí, jde obvykle o osobní zkušenost týmu, který danou implementaci připravuje. A na základě této zkušenosti je pak vydáno rozhodnutí. Poměrně pěkný nestranný článek k této tématice byl v roce 2008 vydán na serveru `http://linux.com` [20]. Osobní zkušenosti autora by pak bylo možno shrnout takto:

- HW RAID vyžaduje rozsáhlé studování dokumentace a ovládání obslužného softwaru.
- Ani relativně drahý modul HW RAID od renomované firmy nemusí znamenat jistotu bezpečí dat.
- Komplexita přidaná SW RAID je minimální.
- Jednotnost přístupu k diagnostice SW RAID a jednotnost jeho administrace velmi zjednodušuje údržbu většího množství strojů.
- Oproti serverům bez RAIDu nebylo pozorováno žádné kritické zpomalení na serverech s RAIDem.

Díky těmto vyjmenovaným a několika dalším zkušenostem jsme se rozhodli pro nasazení SW RAID na všech produkčních serverech. Jiná skupina by díky jiným argumentům mohla dojít k zcela odlišnému závěru.

Velmi zajímavou studii na téma životnosti rotačních mechanických pevných disků vydala v roce *2007* společnost *Google* pod názvem *Failure Trends in a Large Disk Drive Population* [11]. V této studii stojí za povšimnutí dva body:

1. Disk selže s relativně vysokou pravděpodobností v prvních třech měsících fungování. Pokud neselže v tomto čase, je velká pravděpodobnost že bude fungovat až do doby výměny za novější model, či kompletní výměny serveru.

2. Mírně zvýšená provozní teplota pevnému disku téměř nevadí. Ba naopak nízké provozní teploty přinesly kratší životnost mechanických pevných disků.

1.6.2 Failover výpočetního serveru

Jak již bylo probráno v kapitole 1.2 a 1.5 je pro naše výpočetní prostředí velmi důležitá schopnost nepřerušeného běhu s výpadkem jakéhokoliv výpočetního stroje. Výpadky serverů se stávají a nelze jim nikdy stoprocentně zabránit. Výpadek může nastat díky poruše hardwaru, selhání disku[10], výpadku na síťovém segmentu, selhání ze strany aplikací či operačního systému, nebo jde „pouhou“ chybou v konfiguraci serveru.

V případě, že bychom byli odkázáni pouze na jeden výpočetní stroj a došlo by k jeho selhání, pak se aplikace jako celek jevila jako nedostupná. Tomuto nepříjemnému jevu bylo zabráněno předsunutím *load balanceru* před všechny výpočetní servery. Tuto situaci znázorňuje obrázek 1.5. Vlastnosti tohoto řešení jsou následující:

- V případě výpadku kteréhokoliv výpočetního serveru převezme jeho práci automaticky jiný.
- Zátěž je rovnoměrně rozdělena mezi výpočetní servery a nedochází tak k jevu přetížení jednoho, zatímco ostatní nemají nic na práci.
- Namísto několika míst, jejichž výpadkem by byla aplikace nedostupná, vznikl Single Point Of Failure (SPOF) v podobě *load balanceru*. Tato situace samozřejmě není ideální, ale je rozhodně lépe kontrolovatelná.
- Odstávky výpočetních serverů nejsou v tomto prostředí problém. Toto je velmi důležitý fakt z důvodu aktualizace jádra, povýšení distribuce, či výměny disku. Administrátoři budou mít navíc volnější ruce k experimentování pro různá nastavení systému, softwaru či aplikace samotné. Nemusí se totiž bát, že by chybným zásahem do systému ohrozili produkci.

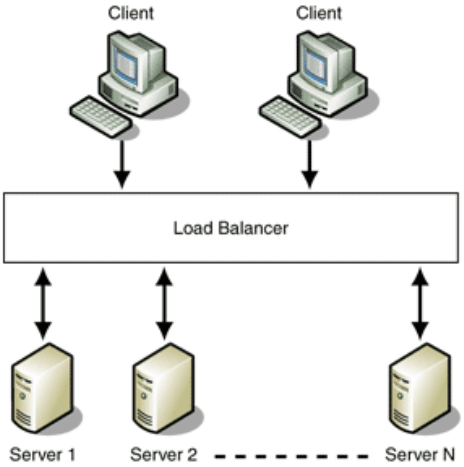
1.6.3 Failover uživatelských dat

Nainstalovat aplikaci na každý výpočetní server je relativně snadná záležitost. Instalace aplikace *lokálně* na každý server je výhodná z několika důvodů. Je zrychleno načítání aplikace do paměti daného výpočetního serveru. Provoz tohoto serveru není pak nijak závislý na jiných serverech. A v neposlední řadě tak vzniká možnost dočasných experimentálních úprav přímo na produkci.[11]

[10] Teoreticky jde jen o speciální případ „poruchy hardwaru“, ale v IT je to tak specifický jev, že jsem ho raději zdůraznil.

[11] Poslední jmenovaný bod není sice tak častý a ani není příliš populární, ale pro nalezení chyb simulovatelných *pouze* na produkčních serverech bývá občas jediným řešením.

Obrázek 1.5: Load balancer



Pokud jde ale o distribuci uživatelských dat, není situace tak jednoduchá. Uživatelskými daty jsou myšleny dokumenty, obrázky, šablony, kaskádové styly a podobné, jež do systému nahrají sami uživatelé či zákazníci, kteří si aplikaci objednali. Nejvýhodnější by samozřejmě bylo mít data uložená na daném výpočetním serveru, jenž se o daného zákazníka stará. Toto bohužel z principu popsaného v kapitole 1.6.3 není možné. Kteréhokoliv zákazníka může obsluhovat kterýkoliv výpočetní server, a proto *není možné* data ukládat lokálně. V původním produkčním řešení byla data distribuována přes protokol NFS.

> Network File System (NFS) je internetový protokol pro vzdálený přístup k souborům přes počítačovou síť. Protokol byl původně vyvinut společností Sun Microsystems v roce 1984, v současné době má jeho další vývoj na starosti organizace Internet Engineering Task Force (IETF). Funguje především nad transportním protokolem UDP, avšak od verze 3 je možné ho provozovat také nad protokolem TCP. V praxi si můžete prostřednictvím NFS klienta připojit disk ze vzdáleného serveru a pracovat s ním jako s lokálním. V prostředí Linuxu se jedná asi o nejpoužívanější protokol pro tyto účely [26].

V původním řešení byl jako server pro uživatelská data použit *file01*, jak je vidět na obrázku A.3. Použití NFS pro *file01* řeší pouze otázku *distribuce*, ale nijak neřeší problém *failoveru* uživatelských dat. Navíc je dobré si uvědomit, že díky faktu, že jsme si o pár řádků výše dokázali „nutnost“ použití load balanceru, jsme fakticky vytvořili další SPOF v podobě *file01*. Pokud nepůjde *file01*, pak budou ovlivněni všichni zákazníci. Pokud bychom měli řešení bez load balanceru (zmiňované výše), pak bychom se do takovéto situace vůbec nedostali.

Failover uživatelských dat je tedy jeden z bodů, jenž je součástí praktické části této práce a bude tedy do detailů rozebráno dále.

1.6.4 Failover databází

Firma používá pro ukládání dat primárně databázové servery "MySQL". Jedná se především o historické rozhodnutí, podpořené předchozími zkušenostmi. Díky vysoké stabilitě databází "MySQL" firma nemusela failover databází řešit. Na obrázku 1.2 je naznačeno rozmístění databázových serverů. Nejedná se o clusterované řešení. Jde tedy pouze o situaci n zákazníků na m strojích, kde $n > m$. Jak už bylo výše řešeno, žádný zákazník svojí databází zatím nepřesáhl výkon jednoho databázového serveru. Není tedy nutné řešit clustering, ale pouze failover. Ten bude rozebrán v kapitole 2.5.

1.6.5 Failover serverů

K selhání hardwaru dochází v IT naprosto běžně a nemusí se jednat jen o pevné disky. Zažili jsme „vyhoření" síťové karty, procesoru, základní desky, ale i routeru a RAID řadiče. Jelikož předmětem podnikání této firmy *není* správa a údržba hardwaru a síťové infrastruktury, byl učiněn logický krok přenechání správy HW infrastruktury jiné firmě. Jedná se o obyčejný server-housing, u kterého jsou pronajaty jejich vlastní servery. Toto řešení má mnoho výhod:

- Firma nemusí vydávat vysoké částky při pořízení nového serveru.
- Firma si *pronajímá* server jako celek, takže provozovatel server housingu je odpovědný za jeho bezvadnost. Tudíž odpadnou starosti a náklady na výměnu komponent serverů.
- Nároky na výkon serverů se postupně zvyšují a dnes zakoupený výkonný server je za rok považován svým výkonem za průměrný. Serverhousingová firma samozřejmě časem zakoupí výkonnější servery, jež za vynaložení minimálních nákladů nahradí ty stávající.
- Vzhledem k obvykle nemalé dojezdové vzdálenosti k serverovně je výhodný fakt, že veškeré servisní zásahy budou řešit technici třetí strany.

1.6.6 Závěr

Nelze říci, že by bylo možné nalézt řešení pro failover obecně. Dokonce nelze ani říci, jak obecně vyřešit failover té či oné služby. Vždy záleží na konkrétním produktu, použitém hardwaru, či využívaných technologiích. Někde se praxí ukáže, že okrajová služba může znefunkčnit systém jako celek, a proto musí být zálohovaná. Jinde naopak může dojít k situaci, že výpadek služby, jež se zdá klíčovou pro běh celého systému, vlastně vůbec nevadí a nijak neohrozí produkci[12]. V textu výše jsou tedy rozebrány hlavní prvky serverového

[12]Toto tvrzení se zdá celkem paradoxní, ale na příkladu dále bude vysvětleno.

prostředí, jenž pro bezvýpadkové fungování aplikace firmy potřebují failover. V textu dále pak bude rozebráno, jak jsme daná řešení aplikovali v praxi.

1.7 Zabezpečení

U rychle se vyvíjejících projektů autoři často vůbec, nebo jen velmi povrchně, dbají na zabezpečení. Je to chyba a obvykle se autorům dříve či později vymstí. Zabezpečení opět může probíhat na několika úrovních. Velmi dobrý postup pro definování potřebných prvků zabezpečení je metoda otázek. Lidé, jenž vytváří prostředí pro běh projektu, se navzájem ptají na jakékoliv otázky, jenž je k zabezpečení napadnou. Sepíší je na papír, seřadí do skupin podle tématicky podobných a pokusí se navrhnout nejlépe jedno opatření, jenž pokryje celou skupinu. Takovými otázkami může být například:

- Jak zajistíme, že se do databáze nebude moci nabourat neoprávněný uživatel přes "brute force" útok?
- Jak zajistíme, že uživatel, jenž zcizí heslo k databázi, nebude moci tohoto hesla využít ke stahování citlivých dat?
- Jak docílíme toho, aby útočník, jenž získal přístup k službám serveru, nemohl pole svého vlivu rozšiřovat?
- Jak zjistíme, že se někdo pokouší služby serveru kompromitovat?
- Jak odhalíme bezpečnostní chyby v naší aplikaci?
- Jak odhalíme bezpečnostní chyby v aplikacích používaných na serverech?
- Jak zabráníme zneužití bezpečnostních chyb běžně používaných Open Source Software (OSS) aplikací?

Toto je samozřejmě pouze ilustrační seznam otázek. Ve skutečnosti bývá daleko delší a vytvářený „za pochodu“. Námi nalezené skupiny otázek stran zabezpečení a jejich řešení budou probrány v dalších kapitolách.

Samostatnou kapitolou jenž, úzce souvisí se zabezpečením, je pak *monitoring*. Mnoho lidí tyto dva pojmy velmi snadno spojí v jediný bod. Dle mého názoru jde o dvě velmi odlišné disciplíny. Osobně bych tyto prvky definoval jako:

Zabezpečení Slouží k ochraně systému, aplikace či dat před *známými* vlivy či útoky. Slouží jako prevence před náhodnými či cílenými pokusy o infiltrování infrastruktury produkce.

Monitoring Slouží k sledování hodnot systému a jeho logů pro odhalení známých i *neznámých* nebezpečí. Monitoring známých problémů lze obvykle definovat jednoduchou sadou pravidel. Monitoring *neznámých* problémů je pak obvykle založen na sledování a analýze odchylek ve standardních hodnotách systému, parsování a analýze systémových a aplikačních logů.

Jednotlivé prvky a řešení monitorování použité při řešení této práce budou probrány v dalších kapitolách.

Kapitola 2

Vylepšení produkce

Hlavní motivací pro vznik této práce bylo rozšiřování aplikace, jenž mělo za následek rozšiřování serverové infrastruktury za úroveň, jež by bylo možné zvládnout dělat jen tak „na koleně“. Tato situace si vyžádala vznik specializované pracovní pozice takzvaného *konfiguračního manažera*, jenž má za úkol dohlížet na veškeré prvky produkce od instalace aplikace, přes softwarovou výbavu až po směrování paketů na síti.

Ve chvíli, kdy konfigurační manažer nastoupí do své práce jako první by se měl seznámit s každým detailem produkce, jenž by ho mohl při práci jen trochu zajímat. Nastavení routovacích tabulek, konfigurace firewallu, použité distribuce OS, verze distribucí OS, způsob instalace aplikace, software nainstalovaný na jednotlivých serverech, software instalovaný mimo balíčkovací systém, nastavení virtuálních strojů, speciální záznamy v */etc/hosts*, nastavení Domain Name Server (DNS) serverů, konfigurace a účel proxy serverů, způsob správy konfigurací, proces aktualizace serverů, proces vzdálené administrace serverů, nastavení monitorování prostředí, rozložení know-how o produkci mezi zaměstnanci firmy, spolehlivost jednotlivých služeb a serverů, známé a neznámé problémy produkce, kritické servery a služby pro běh produkce, krizové kontakty...

Výčet by mohl pokračovat ještě mnohem dále. Pro jinou firmu by byl tento seznam odlišný, ale velmi pravděpodobně by některé body sdílel se seznamem uvedeným výše. Je důležité si tedy uvědomit, že objem znalostí potřebný pro rozumné zapracování do pozice konfiguračního manažera je obrovský a odvíjí se od potřeb dané firmy.

Abychom pronikli do prostředí námi zkoumané firmy a do korespondujících potřeb na pozici konfiguračního manažera, uvedeme si několik zajímavých problémů, na než jsem při práci narazil, a na jejich řešení. Řešení zde uvedená se za dané situace s danými prostředky jevila vždy jako nejvhodnější. Pokusím se vždy i vysvětlit, jaká byla motivace právě toho či onoho řešení.

2.1 Náhodné síťové výpadky

2.1.1 Problém

Firma používá jako monitorovací systém "nagios", který jednoho dne začal zobrazovat rozporuplná hlášení ohledně stavu produkce. Občas nahlásil nedostupnost jednoho konkrétního serveru. Při manuální kontrole docházelo k různorodým výsledkům. V jednu chvíli se daný server jevil jako dostupný, při další kontrole o pár minut později jako nedostupný, občas vznikaly stavy, kdy server sice odpovídal na pokusy o spojení, ale vykazoval velký packet loss. Produkce běžela v pořádku dál bez viditelných problémů.

2.1.2 Prvotní příčina

Jelikož byla produkce v době vzniku tohoto problému řešena *ad hoc*, měli zaměstnanci starající se o instalaci nových zákazníků naučený mechanický postup pro instalaci a zprovoznění Secure Sockets Layer (SSL) certifikátů. Každý SSL certifikát potřebuje vlastní Internet Protocol (IP) adresu[1]. Tento problém je velmi pěkně vysvětlen v manuálových stránkách *Apache*:

> Důvod je velmi technický, a poněkud připomíná známý problém „slepice a vejce“. Protocol SSL vrstvy je pod a obaluje HTTP protokol. Když je zahájeno SSL spojení (HTTPS) modul *Apache/mod_ssl* má za úkol vyjednat parametry SSL komunikace s klientem. Pro toto *mod_ssl* musí konzultovat konfiguraci virtuálního serveru (například musí vyhledat šifrovací sadu, certifikát serveru, atp...). Ale k tomu, aby mohl nalézt správný virtuální server *Apache* musí znát položku *Host* hlavičky HTTP protokolu. K tomu, aby toto vykonal, musí přečíst hlavičku HTTP požadavku. Toto nelze provést před tím, než je dokončena fáze takzvaného SSL handshake, ale tato informace je potřeba k dokončení SSL handshake. Bingo [12]!

Na vstupu do naší infrastruktury (*xen02*[2]) je tedy nastaveno velké množství IP adres. Tyto IP adresy se překládají na IP vnitřní sítě, jak je vidět na obrázku A.4[3]. Z historických důvodů jsou tyto IP adresy ze stejného rozsahu jako používají produkční servery pro komunikaci mezi sebou. Bylo tedy jen otázkou náhody, než se tímto způsobem přiřadí IP adresa již běžícího serveru některému z nově příchozích zákazníků.

Problém takovéto situace je ten, že při správné souhře náhod se i při opakované kontrole může server jevit jako dostupný. Administrátor systému má tak mylný pocit,

[1] Toto tvrzení není zcela přesné. Při použití technologie Server Name Indication (SNI) je možné mít IP adresu jedinou. Z obchodních důvodů ale nebylo možné v době řešení tohoto problému SNI použít.

[2] Viz obrázek A.2

[3] IP adresy jsou pouze ilustrační.

že v nepořádku je monitorovací software (nagios). Toto je vlastně velmi pěkný příklad *souběhu*. Souběh je dle wikipedie definovaný jako:

> Souběh (anglicky race condition) je chyba v systému nebo procesu, ve kterém jsou výsledky nepředvídatelné a závislé na pořadí nebo načasování jednotlivých operací. Souběh může nastat v elektronických systémech (zvláště u logických členů) a v počítačových programech (zejména ve víceúlohových a víceprocesorových systémech) [28].

V našem případě došlo k souběhu na úrovni ARP protokolu. Uvažujme příklad kdy v jedné podsíti existují dva počítače se stejnou IP adresou. Jelikož na jednom segmentu sítě neexistuje mechanismus jenž by takovému stavu zabránil, může se na dané podsíti vyskytovat hned několik variant ARP odpovědí proti jednomu ARP dotazu. Toto jednoduše řečeno povede k nepředvídatelným záznamům v ARP tabulkách různých zařízení, tak jak je možno pro příklad vidět na výpisu 2.1.

```
1 lb01:~ # arp -n
2 Address                HWtype  HWaddress           Flags Mask            Iface
3 10.0.0.11              ether   00:14:3e:a4:86:4d   C                     eth0
4
5 watch01:~ # arp -n
6 Address                HWtype  HWaddress           Flags Mask            Iface
7 10.0.0.11              ether   00:27:4b:c4:16:49   C                     eth0
```

Listing 2.1: Ukázka souběhu protokolu ARP

V konečném důsledku se tak jeden stroj může jevit z jednoho místa jako dostupný a z místa jiného jako nedostupný.

2.1.3 Řešení

Business požadavkem pro tento projekt bylo nalézt řešení rychle i s rizikem toho, že bude jen dočasné. Rozdělení adresního prostoru klientů a serverů tedy nepřipadalo v úvahu. Taková akce by zabrala mnoho času. Vytvoření statické mapy IP adres používaných na produkci zase nemuselo přinést ani dočasné výsledky, neboť by lidskou chybou v takovémto seznamu mohly vzniknout nepřesnosti. Jedinou rozumnou možností se tedy jevilo vytvoření *dynamického* seznamu IP adres. Zaměstnanci instalující nové SSL certifikáty tak mohou v jakýkoliv okamžik zkontrolovat zda nepoužijí již zabranou IP adresu. Navíc se výstup tohoto projektu bude hodit pro dokumentaci produkce. Výstup skenování síťě musí být importován do firemních wiki stránek.

Pro tento projekt jsme bohužel nenalezli žádný hotový software. Museli jsme tedy naimplementovat vlastní řešení. Jako programovací jazyk byl zvolen Python. Pro sbírání dat o aktuálně přiřazených IP adresách byl zvolen protokol Simple Network Management Protocol (SNMP), jenž je na produkci již zaběhnut a využívá se k několika dalším účelům.

Obrázek 2.1: IP SNMP scanner

db03-host

lo	**127.0.0.1** = localhost,
db03	
eth1	
eth0	**77.156.99.41**
br0	**10.109.0.166** = db03-host

ext01

lo	**127.0.0.1** = localhost,
eth0	**10.109.0.99** = ext01

SNMP je v principu jednoduchý síťový protokol umožňující sběr dat z routerů, serverů a jiných zařízení. SNMP je dnes implementováno ve valné většině zařízení a je k dispozic ve většině distribucí. Protokol na základě standardizovaných identifikátorů takzvaných Object Identifier (OID) exportuje různé hodnoty o stavu serveru. Například o: Využití CPU, volném místě na disku, teplotě procesoru, atp... Protokol SNMP je snadno rozšiřitelný o vlastní funkce. Nemusí sloužit pouze k čtení, ale nabízí i možnost ovlivňovat stav a běh daného zařízení. Tato vlastnost je ale využívána jen zřídka.

Ve výsledku byl vytvořen jednoúčelový script procházející seznam známých serverů[4] a na každý server pošle dva SNMP dotazy. Jeden zjišťuje seznam všech *ethernetových* rozhraní serveru a druhý k těmto rozhraním dohledává příslušné IP adresy. Pro pohodlí uživatelů je pak ještě proveden dotaz na reverzní DNS záznam[5]. Výsledek je vrácen jako tabulka ve *wiki* syntaxi. Script je k nahlédnutí v kapitole B.1. Volán je pak z cronu a přes cli rozhraní tracu je jeho výstup importován do wiki. Ukázka výstupu je pak k nahlédnutí na obrázku 2.1.

2.2 Důvěryhodný a užitečný monitorovací nástroj

2.2.1 Problém

Jak již bylo řečeno výše, firma používá k monitorování serverů a služeb "nagios". Takovýto monitorovací nástroj je neodmyslitelnou součástí každé produkce. Není v silách žádného člověka mít neustálý a dokonalý přehled o zdraví každého serveru a každé jeho službě jež je součástí produkce. Monitorovací server dokáže zavčasu upozornit na případné budoucí, či právě vzniklé problémy. Základní předpoklady každého monitorovacího systému jsou:

1. Monitorovací server musí ležet mimo monitorovaný systém. Pokud by došlo k výpadku síťové konektivity, jen těžko by monitorovací nástroj ohlásil problémy, pokud by byl

[4]Tento seznam se mění relativně zřídka.
[5]Reverzní DNS záznam přiřazuje IP adrese doménové jméno

na postižené síti.

2. Monitorovací server nesmí být ovlivňován žádnými vnějšími vlivy jako je například zatížení sítě, na které běží, či vytížení serveru, ze kterého je spuštěn.

3. Musí sledovat všechny známé kritické služby a servery, jenž mohou ovlivnit dostupnost produkce.

4. Monitorovací systém *ideálně* nesmí vyprodukovat *žádné* falešně pozitivní hlášení.

Problém existujícího monitorovacího serveru byl ve všech výše zmíněných bodech. Fyzicky byl umístěn v kanceláři firmy, jejíž servery současně monitoroval. Jelikož v kancelářích firmy probíhá čilý síťový provoz, bylo vyvoláváno relativně velké množství falešně pozitivních poplachů díky síťovým problémům na straně *nagiosu*. Díky této vlastnosti byli po nějaké době administrátoři sledující hlášení nagiosu natolik „imunní“ vůči různým hlášením, že snadno přehlíželi důležitá (pravá) hlášení havárií. A v neposlední řadě díky delší době od poslední úpravy konfigurací *nagiosu*, nesledoval tento všechny potřebné služby.

2.2.2 Řešení

Za velmi zajímavé zjištění během realizace tohoto projektu osobně považuji souvislost mezi vytížením serveru, na kterém běží nagios a počtem falešně pozitivních hlášení. Původní idea byla přenést společně s nagiosem na nový server i cacti[6]. Toto se ale v praxi ukázalo jako nepoužitelné řešení. Cacti svojí přítomností vytěžovalo monitorovací server velmi významným způsobem. Po jeho odsunutí na jiný server byl zredukován počet falešně pozitivních hlášení na cca 1/3.

Po prostudování možných řešení bylo rozhodnuto, že se *nagios* přestěhuje na cloud společnosti Amazon[7]. Důvody pro tuto volbu byly následující:

- Díky elasticitě cloudu je zvýšení výpočetní síly serveru velmi jednoduché. Důležitá vlastnost díky výše zmíněným falešně pozitivním hlášením.

- Známá vysoká spolehlivost této služby přináší jistotu dostupnosti a běhu nagiosu.

- Díky odděleným zónám by výpadek jedné neměl ovlivnit jiné. Navzdory umístění ve stejné geografické lokaci.

- *EC2* byla plánována k bližšímu prozkoumání. Takže byl výhodný fakt, že výstup této elaborace poslouží jako základ pro další práci.

[6]Software pro sběr statistik o hodnotách serveru a jejich zobrazení do grafu.

[7]Přesněji *Amazon Elastic Compute Cloud (Amazon EC2)*. Tato služba bude rozebrána v dalších kapitolách této práce.

Instance nagiosu používaná ve firmě byla již staršího data a ne vše bylo řešeno optimálně. Kromě aktualizace softwaru samotného bylo potřeba provést vyčištění monitorovacích služeb. Velmi mnoho modulů bylo napsaných „ručně“ a využívalo nejrůznějších nesprávných postupů. Jako příklad mohu jmenovat použití Secure Shell (SSH) protokolu pro monitorování služeb, jenž lze nativně monitorovat přes SNMP, použití SNMP pro zjišťování času na serveru ačkoliv každý server poskytuje tento výstup přes Network Time Protocol (NTP), moduly hlídající několik vlastností najednou, které díky špatné implementaci nebyly schopné reportovat více závad *současně*, a další...

Většina těchto problémů jednoduše zmizela použitím *standardního* přístupu pomocí *standardních* SNMP modulů nagiosu. V distribuci *Debian* lze jednoduše nainstalovat pomocí balíku *nagios-snmp-plugins*. Již několikráte zdůrazňovaná vlastnost *standardního* přístupu je dle mého názoru velice důležitá. Standardní modul nagiosu je, na rozdíl od „ručně“ psaného komunitou udržován a opravován. Obvykle mívá rozsáhlou dokumentaci a dohledatelné velké množství příkladů použití. Je velmi malá pravděpodobnost, že by s aktualizací systému přestal fungovat. Pokud by do týmu přibyl další administrátor, bude mít díky výše uvedeným vlastnostem jednodušší seznámení se systémem.

Naopak za velmi nevhodné považuji použití SNMP pro zjišťování systémového času. Tento přístup nijak nepočítal s časem průchodu paketu sítí. Teoreticky se tedy mohl čas zjištěný touto metodou od reálné času hodin serveru dramaticky[8] lišit. Navíc *standardní* modul nagiosu pro zjišťování času přes NTP existuje, tak proč ho nevyužít?

Sledování problémů aplikace

Samostatnou kapitolou pak byla implementace modulu nagiosu/aplikace pro hlášení známých problémů aplikace. Původní řešení bylo založeno na periodickém prohlížení aplikace nagiosem a zkoumání,zda je vše v pořádku. Tento přístup měl několik velmi zásadních vad:

1. Seznam URL, jež musí být nagiosem kontrolován se musel manuálně udržovat/rozšiřovat při změně/přidání zákazníka.

2. Nagios musel mít přístup do aplikace. Pokud kontroloval část, jenž vyžadovala autentikaci/autorizaci, bylo nutné mu ji poskytnout.

3. Ve chvíli, kdy nagios měl za úkol takto periodicky kontrolovat *stovky* instalací, začal svou roli hrát výkon a zpomalení aplikace v důsledku kontrol.

4. Díky velkému množství kontrolovaných instancí aplikace docházelo k prodlevám mezi poruchou a jejím nahlášením.

5. Díky heterogenitě instancí aplikace nebylo úplně jednoduché vytvořit kód, jenž by bezchybně detekoval poruchu.

[8] Prakticky tu mluvíme o desítkách až stovkách milisekund.

Díky těmto a několika dalším problémům bylo rozhodnuto pro změnu kontroly aplikace z *aktivního* módu do *pasivního*. Pokud bude aplikace sama hlásit problémy, pak budou velmi elegantně vyřešeny všechny problémy uvedené výše. Samozřejmě v komunitě okolo nagiosu neexistuje modul, jenž by monitoroval konkrétně naši aplikaci, ale naštěstí existuje *standardní* přístup pro příjem *pasivních* zpráv nagiosem. Toto rozšíření se jmenuje Nagios Service Check Acceptor (NSCA) a je *standardním* balíkem ve většině linuxových distribucích. Jeho fungování je znázorněno na obrázku 2.2. Nagios nedělá žádné aktivní kontroly dané služby. Místo toho se aplikace sama v případě problémů připojí na démona NSCA, jenž nagiosu „podstrčí“ výsledek kontroly. Jednoduchý příklad volání NSCA je vidět v kódu 2.2. Pro správné fungování je potřeba mít nainstalován balík NSCA jak na monitorovacím, tak monitorovaném systému. Příklad konfigurace nagiosu je vidět v kódu 2.3.

```
1  /// Syntax for NSCA message: <hostname>\t<svc_description>\t<return_code>\t<plugin_output
       >
2  /// <hostname> Hostname of the machine where error occured
3  /// <svc_description> Brief description of the problem
4  /// <return_code> 0=OK, 1=WARNING, 2=CRITICAL
5  /// <plugin_output> Description of the problem
6
7  $customer_id = Config::get('customer', 'id');
8  $return_code = 2;
9  $msg = sprintf('lb01\tDB Error\t%d\t%d - db error occured.', $return_code, $customer_id);
10 exec('echo -e "'.$msg.'" | /usr/sbin/send_nsca -H watch01');
```

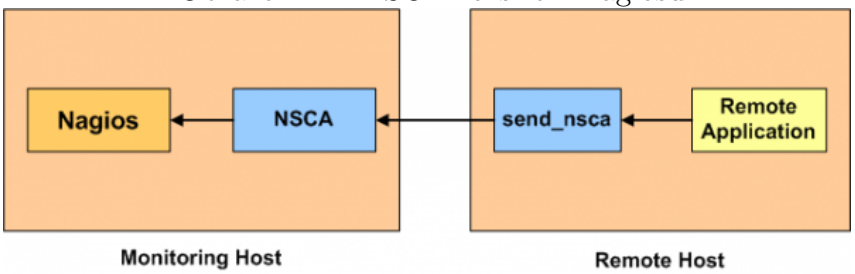
Listing 2.2: Ukázka volání NSCA z PHP

```
1  define service {
2      use                     generic-service
3      host_name               lb01
4      check_command           dummy-ok
5      service_description     DB Error
6      passive_checks_enabled  1
7      normal_check_interval   15
8  }
9
10 define command {
11     command_name  dummy-ok
12     command_line  echo "Automatically restarted"
13 }
```

Listing 2.3: Ukázka konfigurace nagiosu pro hlídání chyb databáze

Za povšimnutí stojí nadefinování příkazu *dummy-ok*, jenž je použit jako *aktivní* kontrola služby *DB Error*, jenž je spouštěna každých 15 minut. Tento příkaz pouze vrací řetězec „Automatically restarted“ a jako návratovou hodnotu vrací nulu. Použitím tohoto příkazu jako aktivní kontroly v službě *DB Error* dojde každých *15 minut* k vynulování případného chybového hlášení. Toto je velmi výhodné, jelikož se žádný z administrátorů o toto nebude muset ručně starat. Současně bylo odhadnuto, že oněch 15 minut je ideální interval vynulování stavu. Pokud by byl interval menší, pak by mohl vzniknout efekt

Obrázek 2.2: NSCA rozšíření nagiosu



„blikání“, kdy by aktivní kontrola vypnula poplach a vzápětí by pasivní kontrola poplach zapnula. Současně je potřeba tento interval mít co nejkratší, aby obsluha nagiosu viděla kritický stav produkce po co nejkratší dobu.

2.3 Zvyšující se nároky na výkon výpočetních serverů

2.3.1 Problém

Zvyšující se počet zákazníků, narůstající komplikovanost aplikace, narůstající data zákazníků, zvyšující se počet uživatelů aplikace, toto byly hlavní důvody postupného snižování výkonu aplikace a zvyšování odezvy systému jako celku. To bylo samozřejmě pro uživatele, zákazníky a v konečném důsledku i pro firmu jako takovou nepříjemné. Vznikl tedy projekt mající za cíl navýšení výkonu systému. V následujících kapitolách se pokusím nastínit zjištěné výsledky, motivace jednotlivých kroků a dopady na následující provedené kroky. Zjištěné závěry jsou sice na míru šité naší situaci, ale obecné poučení si lze vzít pro jakoukoli středně velkou[9] webovou aplikaci.

2.3.2 Projekt Amazon VPC

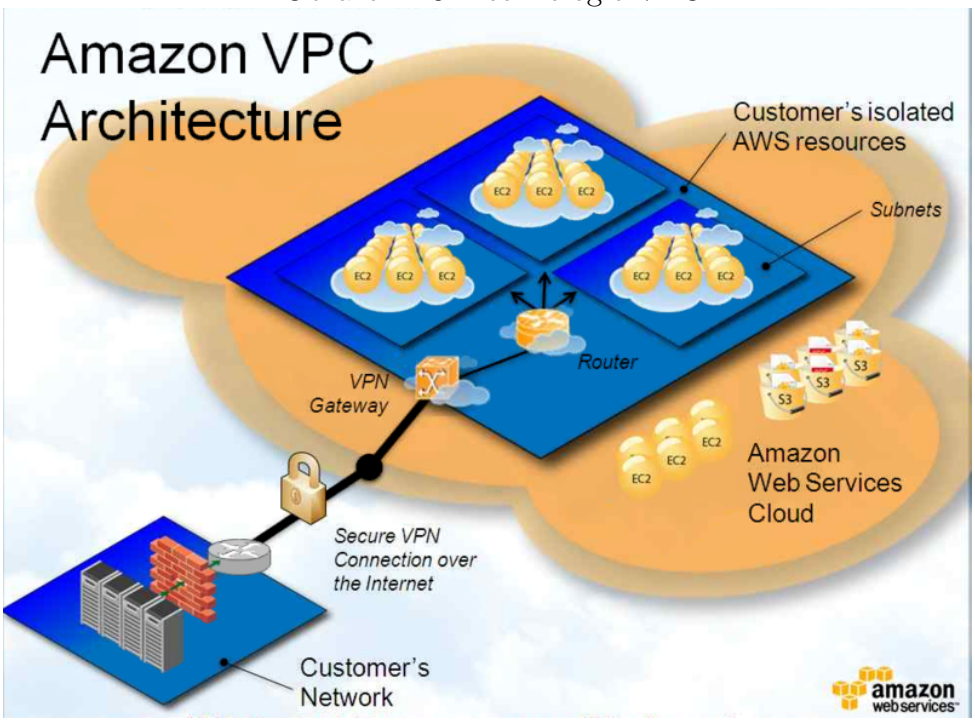
V posledních letech se hojně mluví o výpočtech v takzvaném „cloudu“. Princip je vcelku jednoduchý. Zákazník si pronajme určitý garantovaný výpočetní výkon[10] a neřeší serverovou infrastrukturu, kabeláž a dokonce ani virtualizaci serveru pro přerozdělení výkonu. Díky elasticitě cloudových technologií je velmi snadné zvýšit či snížit výpočetní sílu serveru. Přidání dalších procesorů či paměti je otázka několika málo minut. Toto byla pro firmu samozřejmě velmi zajímavá technologie. Nikdo nedokázal dostatečně věrohodně odhadnout růst aplikace a infrastruktury v příštích několika letech. Technologie umožňující jednoduše „nafouknout“ kterýkoliv ze serverů by nám tedy velmi pomohla.

Pro tento projekt byla zvolena technologie Amazon Web Services (AWS), respektive její části Elastic Compute Cloud (EC2) a Virtual Private Cloud (VPC). AWS je souhrnný název pro všechny služby, jenž společnost Amazon nabízí pro řešení webových projektů.

[9] Stovky instancí aplikace, každá se stovkami až desítkami tisíc uživatelů. Aktivních je v jednu chvíli až 1/1000 z celkového počtu uživatelů.

[10] Obvykle udávaný jako výkon Central Processing Unit (CPU) a množství RAM.

Obrázek 2.3: Technologie VPC



Pod pojmem EC2 se pak skrývá vlastní pronájem výpočetního prostoru na serverech Amazonu. Stará se o manipulaci s výkonem strojů, vytváření nových, správu existujících. Technologie VPC pak umožní rezervování určitého výpočetního výkonu a jeho napojení na naši vlastní serverovou infrastrukturu. Tyto servery jsou s infrastrukturou zákazníka (náší) spojeny přes zabezpečenou linku VPN. Tuto situaci zachycuje obrázek 2.3

Amazon AWS nabízí kromě webového rozhraní také velmi příjemnou sadu nástrojů pro správu virtuálních serverů z příkazové řádky. Díky tomu je možné vytvořit jednoduché skripty, jenž budou například zvyšovat výkon výpočetních serverů v případě potřeby a naopak ho snižovat pro ušetření nákladů. Díky své popularitě jsou tyto nástroje snadno instalovatelné na všechny přední linuxové distribuce. Amazon sám nabízí uživatelskou příručku[3], příručku pro jejich Application Programming Interface (API)[1] i příručku pro práci s cli rozhraním[2].

Popisovat do detailů jednotlivé technologie Amazon AWS je mimo rozsah tohoto dokumentu. Osobně si myslím, že mají široké spektrum uplatnění a nabízí jistý technologický náskok před „standardním“ přístupem pronájmu dedikovaných serverů.

Zadání projektu bylo velmi jednoduché: Připojit technologii Amazon VPC na stávající infrastrukturu a zajistit spuštění dostatečného počtu výpočetních virtuálních strojů v případě, že nastane potřeba. Potřebou bylo nejčastěji myšleno výkonnostní pokrytí špiček. Jelikož je aplikace používána v několika málo časových pásmech, rozložení návštěvnosti produktu přes den má charakteristiku Gaussovy křivky[24], jak je vidět na obrázku A.10, je tedy potřeba násobně vyššího výkonu v poledních hodinách oproti výkonu potřebnému v noci. Dalším momentem jsou skokové nárůsty návštěvnosti, jenž obvykle nastávají po nasazení

nové instance produktu, rozeslání hromadného e-mailu uživatelům, nebo na základě nějaké události uvnitř komunity uživatelů produktu. Nárůst návštěvnosti mezi dvěma po sobě jdoucími dny tak může dosáhnout až dvojnásobku. Tuto situaci ukazuje den *20* a *21* na obrázku A.11. Amazon VPC bylo tedy naše teoretické řešení výše popsaných problémů, a to díky schopnosti technologie poskytnout během několika minut server s desítkami gigabytů operační paměti a desítkami procesorů. Zjednodušené schéma napojení této technologie na naši infrastrukturu je vidět na obrázku A.5. Výhoda využití této technologie místo pronájmu dalších dedikovaných serverů je ta, že můžeme dynamicky reagovat na potřeby systému a platíme pouze za čas, kdy jsou servery využívány.

Implementace

Při implementaci tohoto projektu jsme se setkali s efektem pro nás do té doby nevídaným: Naprostá degradace výkonu systému jako celku při zvýšení Round Trip Time (RTT). Dle wikipedie je RTT definováno jako:

> V telekomunikačních systémech je jako „round-trip delay time“ nebo „round-trip time“ označován časový interval potřebný pro odeslání a příjem potvrzení signálu. V kontextu počítačových sítí je tímto signálem obvykle označován datový paket a RTT je bráno jako takzvaný „čas pingu“. Uživatel počítačové sítě může zjistit RTT použitím příkazu *ping* [29].

RTT mezi servery stávající infrastruktury se pohybuje v rozsahu $0.1ms$ až $0.15ms$. Po připojení testovacího serveru VPC jsme zjistili, že ping ze serverů Amazonu na servery stávající infrastruktury je přibližně $30ms$. Což je přibližně $300\times$ násobek původní hodnoty. Vysoký čas RTT obvykle nevadí při přenosu velkého objemu dat, který je uskutečněn v jedné dávce. Tehdy rozhoduje hlavně kapacita přenosové cesty. Bohužel náš případ byl ale zcela odlišný. V našem systému probíhá pro zobrazení každé webové stránky množství (řádově stovky až tisíce) "MySQL" dotazů. Dále zde jsou řádově stovky dotazů na NFS server. V potaz je třeba brát také statické zdroje, jenž sebou každá dynamická stránka přináší[11], a jenž vytváří další HTTP dotazy.

Závěr

Závěr tohoto projektu byl tedy takový, že navzdory několikanásobnému navýšení síly výpočetních serverů se celková doba odezvy systému zvýšila. Měřením bylo zjištěno, že doba odezvy se zvýšila na přibližně *deseti násobek*. Tento výsledek je vlastně velice jednoduše matematicky dokazatelný. Pokud budeme uvažovat počet "MySQL" dotazů jako $a = 500$, počet NFS dotazů jako $b = 100$, počet statických zdrojů jako $c = 10$, dobu

[11] Obrázky, kaskádové styly, soubory JavaScriptu, a podobné...

RTT mezi Amazonem a stávající infrastrukturou jako $R = 30ms$, pak můžeme výslednou *minimální* dobu t potřebnou k načtení jedné kompletní webové stránky odhadnou jako:

$$t = (a + b + c) * R$$

$$t = (500 + 100 + 10) * 30$$

$$t = 18300ms$$

$$t = 18,3s$$

Výsledný čas bude samozřejmě ještě vyšší, jelikož daný vzorec nepočítá s dobou práce na jednotlivých serverech.

Tento projekt byl tedy *neúspěšný* a od používání takto navrženého systému bylo nakonec upuštěno. Projekt přinesl jeden důležitý výstup, a to sice poznání, že nemáme nalezena úzká hrdla systému a nemáme prozkoumáno chování systému při změně parametrů (ping, odezva disků, odezva NFS serveru, propustnost sítě, atp...).

2.4 Projekt „zóna 2“

Výstupy a zjištění z neúspěšného projektu minulé kapitoly posloužily jako základ projektu dalšího. Jak už bylo řečeno RTT hraje v našem prostředí velmi významnou roli. Každý server má dvě síťová rozhraní. První o rychlosti *100 Mbit*[12] je připojeno na routery server-housingové společnosti a slouží pro připojení stroje do internetu. Druhé o rychlosti *1 Gbit*[13] slouží pro komunikaci serverů mezi sebou. Bohužel díky počtu serverů v naší infrastruktuře jsme dosáhli technologického maxima připojených strojů v jednom Gigabitovém switchi pro rozhraní *eth1*, tedy pro komunikaci po vnitřní síti. Přidávání dalších výpočetních či databázových serverů tudíž nebylo jednoduše možné. Přidání dalších serverů samozřejmě možné je, ale byly by spojeny „pouze“ *100 Mbit* sítí na úrovni rozhraní *eth0*. Toto není žádoucí situace z důvodu zvýšení doby odezvy mezi jednotlivými servery. Zvyšování počtu výpočetních a databázových serverů je v naší situaci nevyhnutelné.

A tak bylo potřeba vymyslet způsob, jak dovolit rozdělení produkce do několika víceméně nezávislých zón. V rámci zóny by komunikace mezi servery probíhala rychlostí *1 Gbit* a jednotlivé zóny mezi sebou by pak komunikovaly rychlostí *100 Mbit*, samozřejmě za předpokladu co nejmenšího výkonnostního dopadu při vzájemné nevyhnutelné komunikaci přes rozhraní *eth0*.

[12] Pro jednodušší orientaci jej budu označovat jako eth0
[13] Pro jednodušší orientaci jej budu označovat jako eth1

2.4.1 Problém vzdálenosti mezi databázovými a webovými servery

Jak již bylo naznačeno v minulé kapitole, výpočetní servery provádějí velká množství dotazů do databází. Je tedy velmi důležité zajistit, aby vzdálenost (ping) mezi webovým serverem a databázovým serverem byla co nejmenší. Z principu fungování našeho load-balanceru toto není problém. Load-balancer lze nastavit tak, že dotazy na určitý *hostname*, nebo-li doménu bude směřovat vždy jen na určitou skupinu výpočetních serverů. Tímto nastavením a rozmístěním zákaznických databází na odpovídající databázové stroje pak lze dosáhnout toho, že daného zákazníka budou vždy obsluhovat pouze ty výpočetní stroje, jenž mají blízko k databázovému stroji daného zákazníka. Tuto situaci zachycuje obrázek A.6. Ukazuje situaci při přidání dvou nových výpočetních serverů (web04, web05) a jednoho databázového serveru (db04). Dotazy na doménu `www.X.cz` a `www.Y.cz` jsou load-balancerem poslány na jeden z výpočetních serverů zóny 1 (*web02*, či *web03*). Tyto se pak dotazují databázových strojů *db01* (pro URL `www.X.cz`), nebo *db03* (pro URL `www.Y.cz`). Pro zónu 2 je pak postup obdobný.

Tohoto chování lze dosáhnout vytvořením konfiguračních souborů apache s obsahem uvedeným ve zdrojovém kódu 2.4 a 2.5.

```
1 <Proxy balancer://portalall>
2     BalancerMember http://web02:80 route=web02
3     BalancerMember http://web03:80 route=web03
4 </Proxy>
```

Listing 2.4: /etc/apache2/zone01.cnf

```
1 <Proxy balancer://portalall>
2     BalancerMember http://web04:80 route=web04
3     BalancerMember http://web05:80 route=web05
4 </Proxy>
```

Listing 2.5: /etc/apache2/zone02.cnf

Tyto definice se pak jednoduše použijí v definicích virtual hostů jednotlivých domén jak ukazuje zdrojový kód 2.6. Použití jiného konfiguračního souboru zajišťuje direktiva *Include*.

```
1  <VirtualHost 10.0.0.1:80>
2          ServerName www.X.cz
3          Include zone01.cnf
4  </VirtualHost>
5
6  <VirtualHost 10.0.0.2:80>
7          ServerName www.Y.cz
8          Include zone01.cnf
9  </VirtualHost>
10
11 <VirtualHost 10.0.0.3:80>
12         ServerName www.Z.cz
13         Include zone02.cnf
14 </VirtualHost>
```

Listing 2.6: Ukázková definice virtual hostů

2.4.2 Problém vzdáleného úložiště uživatelských souborů

Pro naše prostředí platí, že úložiště uživatelských souborů musí být kdykoli dostupné pro každý výpočetní server. Se zavedením zóny 2 se námi používaná technologie NFS ukázala jako nedostačující. Výpočetní servery zóny 2 mají „daleko“ na server uživatelských dat *file01*. Vytvářet nový souborový server (*file02*), jenž by uchovával uživatelské soubory pouze pro zákazníky zóny 2 se nám nezdálo příliš vhodné.

- Vzrostla by komplikovanost produkce. Bylo by nutné předělat logiku dlouho běžících úloh na serveru *process01*, které by musely brát v potaz zónu, z které daný zákazník běží.
- Přesun zákazníků mezi zónami by znamenal výpadek aplikace pro všechny právě přesouvané zákazníky. V době kopírování dat mezi *file01* a *file02* musí být aplikace vypnutá, jinak by mohlo dojít k nekonzistenci uživatelských dat.
- Stále by nebyl vyřešen problém failoveru uživatelských dat nastíněný v kapitole 1.6.3.

Hledali jsme technologii pro vzdálený přístup k souborům na lokální síti, se schopností replikace dat mezi více servery. Možností nepřerušeného běhu i za předpokladu, že některý ze serverů vypadne. Technologie musí být schopna běžet i na starším linuxovém jádru, konkrétně *2.6.22*. Nesmí vyžadovat žádné speciality jako patchování jádra či kompilaci jakéhokoliv softwaru ze zdrojových kódů. Technologie by měla být z administrátorského pohledu co nejjednodušší.

Výčet distribuovaných souborových systémů je v dnešní době relativně široký: DRDB, GFS, OCFS, GlusterFS, Lustre, OpenAFS... Po pečlivém uvážení jsme vybrali GlusterFS. GlusterFS je dle slov autorů:

> GlusterFS je open source, clusterový souborový systém, schopný pojmout data v řádech petabajtů a obsluhovat tisíce klientů. Servery zapojené do GlusterFS tvoří stavební bloky přes Infiniband RDMA a/nebo TCP/IP propojení. Sdílí přitom diskový prostor, operační paměť pro správu dat v jediném globálním jmenném prostoru. GlusterFS je založeno na modulárním designu v uživatelském prostoru a dokáže přinést výjimečný výkon pro různé druhy úloh [8].

GlusterFS používá výraz *volume* pro diskový oddíl, jenž vytváří a poskytuje klientům pro čtení/zápis. Výraz *brick* je pak použit pro jednotlivé stavební komponenty *volume*. Technologii GlusterFS jsme plánovali použít pro vytvoření replikovaného *volume* s tím, že každý server bude poskytovat právě jeden *brick*.

V našem případě byly hlavní důvody pro výběr právě této technologie následující:

- GlusterFS je založený na Filesystem in userspace (FUSE). Instalace na starší jádro (kernel) není problém.
- Web vývojářů nabízí jak *deb*, tak *rpm* balíčky[14].
- GlusterFS předpokládá pouze POSIX souborový systém. Na jeho konkrétní implementaci už nezáleží. Je možné používat na jednom serveru *ext3* a na jiném *ext4*
- Z předchozího bodu nepřímo vyplývá snadná integrace do již existujícího systému, kdy je jako GlusterFS oddíl označen pouze nějaký *adresář* a nikoliv celý souborový systém.
- GlusterFS je *velmi* snadno nastavitelný a ovladatelný[15].
- Technologie nabízí vlastní protokol pro přístup k souborům. Při použití tohoto protokolu automaticky získáme failover.
- GlusterFS automaticky nabízí přístup k souborům také přes NFS. Výhodné pro zachování případné nutné zpětné kompatibility.
- Je možné provést počáteční synchronizaci dat replikovaných *brick* a následně vytvořit *volume* bez jakýchkoliv problémů s počáteční synchronizací velkého objemu dat.

Za nevýhody zjištěné *praktickým používáním* považuji:

1. Velmi laxní přístup autorů k dokumentaci. Odpovědi na většinu otázek jsem získal až při komunikaci přímo s autory / uživately GlusterFS.
2. GlusterFS ukládá data o replikaci přímo do rozšířených atributů daných souborů, čímž se o cca 1/4 zvětší velikost zabraného místa na disku.
3. V případě, že by někdo zapsal data přímo do fyzického souboru, jenž je použit v GlusterFS *volume*, pak dojde k nedefinovanému stavu a obsah souboru je prakticky ztracen.

Princip replikace popsaný v dokumentaci GlusterFS udává „potřebu kontrolovat aktuálnost souborů na všech *brick* při určitém typu souborových operací“ [7]. Toto samozřejmě přináší otázku optimálního rozmístění serverů pro jednotlivé *brick*. Problémem je zde slabá konektivita mezi zónami. Pokud bychom umístili oba *brick* do jedné zóny, výrazně snížíme odezvu v této zóně, ale úměrně tomu zvýšíme odezvu v druhé zóně. Na druhou stranu umístění *brick* do různých zón může znamenat mírné zvýšení odezvy pro obě zóny. Testy, při kterých jsme přenášeli balík různě velikých souborů, a jejichž výsledky jsou shrnuty

[14]Na produkci je použito *OpenSUSE* (rpm) a *Debian* (deb).
[15]Toto tvrzení prohlašuji pouze o GlusterFS verze *3.1 a vyšší*

Zdrojový server	Typ síťového připojení	Protokol připojení klienta	Čas
file01	100 Mbit	NFS	5m 55s
file02	100 Mbit	NFS	8m 14s
file01	100 Mbit	GlusterFS, 1 brick	5m 32s
file02	100 Mbit	GlusterFS, 1 brick	4m 43s
file01	1 Gbit	GlusterFS, 1 brick	0m 17s
file01 + file02	1 Gbit + 100 Mbit	GlusterFS, 2 bricks	2m 36s
file01	1 Gbit	NFS	0m 15s
file02	1 Gbit	NFS	0m 22s

Tabulka 2.1: Výkon GlusterFS (přenos 1,5 GB různě velikých souborů)

v tabulce 2.1 tuto domněnku potvrzují. Rozhodli jsme se pro řešení, kdy každá zóna obsahuje jeden brick. Technologicky pro nás bylo daleko snadněji realizovatelné a preferování vysokého výkonu jen pro určité zákazníky pro nás nebylo přijatelné.

Počáteční synchronizace

V případě, že máte již existující uživatelská data, je možné provést jejich počáteční synchronizaci mezi jednotlivými *brick*. Tím zamezíte obrovskému přetížení serverů, jenž se budou snažit replikovaný *volume* dostat do konzistentního stavu pro všechny *brick*. Pro počáteční synchronizaci byl v našem případě použit program rsync. Objem dat byl $\sim 200\,GB$. V případě použití *100 Mbit* sítě by tedy přenos měl trvat cca *4 hodiny 30 minut*. Celková doba počáteční synchronizace se ale vyšplhala na přibližně *20 hodin*. Na vině je *velké* množství *malých* souborů. Při pokusu s několika málo velkými soubory[16] byla rychlost přenosu prakticky limitována pouze rychlostí síťového rozhraní. Následný pokus synchronizace *rozdílu* dat mezi servery trval přibližně *4 hodiny*.

Výše popsané zjištění bylo samozřejmě velmi nepříjemné. Teoreticky bychom měli vypnout produkci pro všechny zákazníky, provést částečnou synchronizaci a poté opět produkci zapnout. Toto by znamenalo výpadek přes *4 hodiny* pro všechny zákazníky, což samozřejmě nebylo přijatelné řešení. Po delší době strávené nad tímto problémem jsme dospěli ke kompromisu. Ono obrovské množství malých souborů bylo v naprosté většině součástí dočasného adresáře. Do tohoto adresáře si naše aplikace například ukládá: výsledky vyhledávání, jež obsahují velkou množinu dat, zmenšené náhledy obrázků, pdf soubory obsahující exporty dat... Tento adresář tedy obsahuje pouze data, jenž je možné kdykoliv vygenerovat znovu. Testy ukázaly, že synchronizace bez adresáře dočasných souborů potrvá **20 minut**. Tento výsledek samozřejmě není dokonalý, ale pro náš případ byl dostačující. Proto došlo k rozhodnutí provést počáteční synchronizaci následujícím způsobem:

1. Nejdříve bude provedena úplná rozdílová synchronizace. V tuto chvíli bude produkce

[16] Řádově desítky megabajtů na soubor.

stále běžet.

2. Později bude produkce odstavena a bude provedena rozdílová synchronizace, ale s vynecháním *adresáře pro dočasné soubory*.

Skript pro synchronizaci zmíněnou v bodu *2* je vidět ve zdrojovém kódu 2.7. Za povšimnutí zde stojí způsob napojení výstupu programu *find* na vstup programu *rsync* pro vynechání adresářů dočasných souborů. Toto řešení se experimentálně ukázalo jako nejefektivnější.

Takto nám tedy vznikl nový server pro uživatelská data. Díky napojení klientů protokolem *GlusterFS* namísto původního NFS jsme získali schopnost *failoveru*.

```
1  #!/bin/bash
2
3  # This is our source base path which we want to synchronize
4  BASE='/srv/nfs/data1'
5  # Get all first level sub-folders
6  WD=`ls --color=none ${BASE}`
7  # Record when we started
8  date | tee /tmp/start.txt
9  # For better overview of "what is happening", sync first-level subfolders one at a time
10 for i in $WD; do
11     # Show us what subfolder are we synchronizing
12     echo $i
13     # Go to this subfolder
14     cd ${BASE}/$i
15     # Make fast sync
16     # --delete -az = make perfect copy
17     # -i           = show what was changed
18     # --files-from = provide list of file which should be synchronized
19     # -prune       = when find mathces the path ("./custom/*/temp") it will immediatelly
           stop exploring it and any subdirectory
20     RSYNC_PASSWORD='[password]' time rsync --delete -azi --files-from=<( find . -path "./
           custom/*/temp" -prune -o -print ) . rsync://rsync@file02::/volume01/$i
21 done
22 # Record when we finished
23 date | tee /tmp/stop.txt
```

Listing 2.7: Skript pro částečnou rychlou synchronizaci file serverů

Závěr

Rozšířením produkce o *zónu 2* jsme získali potřebný výpočetní výkon. Polední špičky byly částečně pokryty, takže odezva produkce byla méně ovlivňována počtem právě připojených uživatelů. Současně jsme získali failover serveru pro uživatelská data.

Bohužel jsme tímto projektem nedosáhli požadovaného snížení odezvy produkce jako celku. Pro zvýšení výkonu produkce tedy musíme hledat jinou cestu. V budoucnu bude potřeba vyřešit problém nízké přenosové rychlosti sítě mezi zónami (*100 Mbit*), přes kterou nyní musí téci uživatelská data pomocí technologie *GlusterFS*.

Výsledné použití technologie *GlusterFS* v naší infrastruktuře je vidět na obrázku A.7. Replikace mezi servery je naznačena vybarvenou fialovou šipkou. Napojení klientů je

naznačeno prázdnou fialovou šipkou. Z principu fungování tohoto protokolu se klienti připojují k oběma serverům zároveň. Šipka pouze naznačuje server, k němuž se klient pokusí připojit jako první.

2.5 Failover databází

2.5.1 Problém

Každý server může selhat. Může se jednat o selhání disku, špatné nastavení serveru, selhání síťové konektivity, výpadek proudu, jiné selhání hardwaru serveru, softwarové poškození dat na disku, atp... Možností je mnoho. Je proto dobré vždy myslet na záložní řešení pro případ výpadku serveru. Poměrně speciálním případem je failover v případě databázového serveru.

2.5.2 Řešení

Zálohu databázového serveru pro případ jeho nenadálého selhání jsme se rozhodli řešit "MySQL" *master - slave* replikací. Nově vzniklý databázový stroj *db04* je replikován na *db04-slave*. Díky nízké náročnosti replikací jsme si mohli dovolit zálohovací stroje virtualizovat. Server *db04-slave* běží na *slave01-host*, jak je vidět na obrázku A.7. Tímto firma šetří náklady. Tento případ je přesně ten, ve kterém je *správné* a *výhodné* použít virtualizaci serverů. Díky principu replikací v softwaru "MySQL" nevadí, pokud *zálohovací* server na nějakou dobu vypadne.

V případě, že by master server (*db04*) byl nenávratně poškozen, je možné pouhým nastavením *hostname* na DNS serveru a vypnutím replikací na zálohovacím serveru produkci opět plně zprovoznit.

2.5.3 Závěr

Námi zvolené řešení dle standardních definic failoveru vlastně failoverem vůbec není. Některé úkony musí být provedeny manuálně. Nicméně dodává celé produkci stabilitu a možnost rychlého zotavení v případě nouze. Tato část zlepšování produkční infrastruktury je stále ještě v rozvoji.

2.6 Memcached

2.6.1 Problém

Na naší produkci existují určitá data, která jsou velmi často čtena výpočetními servery a musí být pro všechny výpočetní servery stejná. Tato data musí být jednou za čas

hromadně, ve skupinách, invalidována. Tato data mohou být smazána, aniž by došlo k nějakému vážnému problému.

Původní přístup byl ukládat tato data na NFS. NFS svým principem vyhovovalo všem výše uvedeným podmínkám. Tento přístup ale měl jisté nevýhody. Přístup na NFS není v našem prostředí dostatečně rychlý. V případě špatně zvolené struktury se v jednom adresáři mohlo vyskytnout statisíce souborů. Přístup k takovému adresáři přes NFS je pak ještě mnohem pomalejší. Hromadná invalidace záznamů také není nejjednodušší.

2.6.2 Řešení

Pro řešení tohoto projektu jsme se rozhodli nainstalovat sdílený memcached server, jež by využívali všechny výpočetní servery. Software memcached je autory projektu popisován jako:

> Zdarma a open-source, vysoce výkonný, distribuovaný kešovací systém, jež je naprosto obecný, vyvíjený pro použití k zrychlení dynamických webových aplikací zmírňováním databázové zátěže.
>
> Memcached je paměťové úložiště hodnot typu *klíč-hodnota* pro ukládání malých, obecných dat (řetězce, objekty). Primárně určené pro výsledky databázových volání, volání API či celých webových stránek [10].

Memcached má implementované API ve velkém množství jazyků a je podporován jako modul velkou množinou softwaru.

Jedním z míst, kde lze memcached s výhodou použít jsou *session*. *Session* je úložiště informací o uživateli, jenž se vztahuje k danému sezení (přihlášení). Tyto informace z důvodů failoveru musí být k dispozici na všech výpočetních serverech. Současně jsou potřeba při každém načtení webové stránky. Software *PHP* má zabudovanou podporu pro ukládání *session* na memcached server. Stačí pouze pozměnit konfiguraci Hypertext Preprocessor (PHP) tak, jak je vidět na zdrojovém kódu 2.8

```
1 session.save_handler = memcache
2 session.save_path    = "tcp://10.0.0.14:11211"
```

Listing 2.8: Nastavení PHP pro ukládání session na memcached server

Dalším místem, kde se v naší aplikaci nechá memcached použít jsou zákaznické konfigurace. Aplikace před spuštěním vlastního kódu stránky, musí zjistit jakého zákazníka obsluhuje a načíst si všechny jeho konfigurační soubory. Konfigurační soubory aplikace potřebuje při každém svém běhu. Implementaci ukládání uživatelských konfigurací do memcached zde probírat nebudu. Zajímavá je ale disciplína hromadné invalidace dat. Data uložená do memcached nesmí být považována za trvalá. Proto aplikace funguje tak, že se nejprve podívá do memcached, zda obsahuje záznam o nastavení uživatelů. Pokud ho nenalezne, pak si příslušná data načte ze sdíleného diskového prostoru (NFS) a následně

uloží tato data do memcached. Při příštím běhu jsou tak data již k dispozici v rychlém úložišti.

Problém ale nastává ve chvíli, kdy chceme data v memcached invalidovat. Tento software nenabízí možnost práce přes zástupné znaky. Museli bychom tedy ručně definovat sadu dat, jež se má invalidovat. To není hezké řešení. Oficiálně doporučovaný postup v takovéto situaci je předřadit před každý klíč nějaký identifikátor. Například, pokud by původní klíč byl *uzivatel_a* a identifikátr *1234*, pak reálně použitý klíč bude *1234_uzivatel_a*. Dokud bude identifikátor stejný, budou načítána stejná data. Ve chvíli, kdy se identifikátor změní, nebudou žádná data načtena a aplikace uloží do memcached data nová. Tento identifikátor ukládáme na NFS a v případě potřeby ho umíme změnit. Pro zajištění unikátnosti identifikátoru jsme použili číslo revize v subversion (svn). Soubory uživatelského nastavení se na produkci dostávají pouze přes svn. Je tedy jasně dané, že revize z svn dostatečně jedinečně určuje námi požadovaný identifikátor.

2.6.3 Závěr

Prozatím se nám podařilo úspěšně implementovat ukládání *session* a zákaznických nastavení na memcached server. Odhadované zrychlení odezvy systému jako celku nepřesáhlo *5%*. Což není mnoho. Tento projekt přesto přinesl velmi zajímavý výstup. A to zaměření pozornosti na technologii memcached a její propojení v námi používaném PHP frameworku.

2.7 chat01

2.7.1 Problém

Aplikace portálu obsahuje modul pro Instant messaging (IM). Uživatelé si mohou založit diskusní místnosti a v reálném čase zde komunikovat. Pro zjišťování existence nových zpráv uživatele je použito periodické posílání Asynchronous JavaScript and XML (AJAX) dotazů na server. Při bližším prozkoumání jsme zjistili, že tento typ požadavků tvoří přibližně *70%* z celkového počtu dotazů. Nepodařilo se nám předem zjistit, jak velkou část zátěže má tento typ požadavků na svědomí. Výpočty v takto komplikovaném prostředí nebývají moc spolehlivé.

2.7.2 Řešení

Místo odhadování zátěže, bylo rozhodnuto, o provedení experimentu jež měl odklonit tento typ požadavků mimo produkci. Důležitými body v tomto projektu jsou:

- Pro celý systém kontroly nových zpráv byl vyhrazen vlastní server. Nazván byl *chat01* a lze ho vidět na obrázku A.7.

- Původní dotaz šel přes *lb01* na některý z výpočetních serverů. Nyní tedy nebudou zatěžovány ani výpočetní servery a ani load balancer. Dotazy jdou přímo na server *chat01*.
- Pro co největší separaci *chat01* od zbytku produkce bylo nutné vymyslet postup kontroly nových zpráv, jenž by nevyžadoval přístup do databáze. Detaily implementace zde rozebírat nebudu.

2.7.3 Závěr

Díky nasazení serveru *chat01* se snížila zátěž serverů při špičce o přibližně *30%*. Což je dle mého názoru vynikající výsledek. Abych nezkresloval výsledky vlastním výkladem, musím poznamenat, že tento projekt neměl *žádný*, nebo jen minimální vliv na průměrnou dobu odezvy mimo špičku. Lze tedy říci, že se nám podařilo stabilizovat průměrnou dobu odezvy aplikace tak, aby byla méně náchylná na počet aktuálně připojených uživatelů.

2.8 Zjednodušení definic virtual hostů

2.8.1 Problém

Virtual host, nebo-li virtuální server lze považovat za základní jednotku v které se při konfiguraci apache pohybujeme. Každý virtual host představuje jednu, nebo více domén, které na serveru běží. Prostředí virtual hostů produkce je v našem případě převážně homogenní. Díky absenci koncepčního přístupu k vytváření virtual hostů, v nich nebyl původně žádný pořádek. Zvykem bylo kopírovat již existující definice a přepsáním domény vytvořit definici novou. Tento přístup naprosto znemožňoval jakékoliv úpravy, které by se měly dotknout všech zákazníků naráz.

```
1  # Macro to define per-customer configuration. Example:
2  # $customer - Name of custom folder
3  # $version  - Application version (v44svncustom, v47-stable, etc..)
4  <Macro IWCustomer $customer $version>
5  # custom specific resources
6      Use IW_oldresource_3 $version $customer
7
8  # send to full integration
9      RewriteCond %{REQUEST_URI} !.*\.(js|ico|gif|jpg|png|css|html|xml|txt|swf)
10     RewriteCond %{REQUEST_URI} ^/$customer/.*
11     RewriteRule !\.(js|ico|gif|jpg|png|css|html|xml|txt|swf)$ %{DOCUMENT_ROOT}/$version/
           portal/core/public/index.php [L,NS]
12
13     Use IWLib_full $version
14 </Macro>
15
16 <Macro IW_oldresource_3 $version $customer>
17 # old portal resources such as images and scripts
18     RewriteCond %{REQUEST_URI} ^/$customer/.*
```

```
19      RewriteRule /([^/]*)/(.*)\.(js|ico|gif|jpg|png|css|html|xml|txt|swf)$ %{DOCUMENT_ROOT
            }/$version/application/$2.$3 [L]
20 </Macro>
21
22 <Macro IWLib_full $version>
23 # dojo library
24      RewriteCond %{REQUEST_URI} [^/]*/dojo-lib/.*
25      RewriteRule [^/]*/dojo-lib/(.*) %{DOCUMENT_ROOT}/$version/lib/dojo/actual/$1 [L]
26 # fckeditor library
27      RewriteCond %{REQUEST_URI} [^/]*/fckeditor-lib/.*
28      RewriteRule [^/]*/fckeditor-lib/(.*) %{DOCUMENT_ROOT}/$version/lib/fckeditor/actual/
            $1 [L]
29 </Macro>
30
31 Use IWCustomer customer_a       application-47stable
32 Use IWCustomer customer_b       application44
33 Use IWCustomer customer_c       application-pre46
34 Use IWCustomer customer_d       application44
```

Listing 2.9: mod macro

2.8.2 Řešení

Vznikl proto projekt „zjednodušení definic virtual hostů“. Základní výrazové prostředky konfigurací apache nedovolují jednoduše obsáhnout konfigurace naší produkce bez zbytečného opakování. Použili jsme proto modul apache *mod_macro*, který jak již název napovídá, dovoluje psaní maker v konfiguracích apache. Existují i složitější moduly, umožňující používání cyklů, proměnných, funkcí, atp... Takovéto moduly jsme ale nepoužili. Chtěli jsme se vyhnout přílišné komplikaci definic virtual hostů. Bylo by sice sympatické, že by se definice stovek zákazníků vešly na několik málo řádků. Bylo by nám to ale ve výsledku k ničemu, pokud bychom se v nich nedokázali jednoduše orientovat. Tuto myšlenku koneckonců prosazuje i sám autor *mod_macro* na domovské stránce tohoto projektu:

> ... In my opinion, this stuff MUST NOT grow to a full language.
>
> It should just grow to what is needed to manage configuration files simply and elegantly, and to avoid unnecessary copy-pastes. I may consider adding a foreach repetition section, if proven useful. I may also consider reimplementing a working Include since the current one does not work properly (bug #3169/#3578).
>
> But no while, no arithmetics, no variables, no recursion, no goto, no sql, no extended regexpr... Never: I'll stick to the KISS Principle. (Keep It Simple, Stupid!) [6]

Ukázka použití *mod_macro* je vidět na zdrojovém kódu 2.9. Opět se jedná pouze o ukázku. Definice zde uvedená není shodná s tou, jež je používána na naší produkci.

Modul *mod_macro* umí pouze jednu věc. Administrátor si nadefinuje takzvaná makra, jež jsou uzavřena v párovém tagu `<Macro>`. Otevírací tag může přebírat parametry. Obsah těchto maker je pak předložen programu apache vždy, když je makro zavoláno pomocí příkazu `Use`. Případné parametry *foo* a *bar* se pak objeví v těle makra jako *$par_a*, resp. *$par_b*. Tělo makra může obsahovat jiné makro.

2.8.3 Závěr

- Díky modulu *mod_macro* jsme dokázali snížit počet řádek virtual hostů na *desetinu*. Modul ještě není použit na všech místech, kde by to bylo možné. Předpokládáme, že se můžeme dostat až na *5%* původní délky definic virtual hostů.
- Velmi se zlepšila přehlednost konfiguračních souborů.
- Díky jednoduchosti použití *mod_macro* je mnohem menší prostor pro chyby v definicích virtual hostů.
- Nasazení této technologie umožnilo započetí refactoringu virtual hostů. Díky tomu, že jsou definice uvedeny pouze jednou, bez opakování, je možné snadno dělat úpravy v aplikaci, které vyžadují globální změny virtual hostů.

Kapitola 3

Systém pro automatickou správu konfigurací serverů

Z témat probíraných dříve v této práci a zejména z každodenních potřeb při interakci s produkcí vznikl následující seznam požadavků na správu produkčního prostředí:

Dokumentace Produkce vznikala dlouhou dobu ad-hoc bez dlouhodobého konceptu, za účasti různých administrátorů. Dokumentace byla tedy omezena na několik málo wiki stránek, které ke všemu nebyly dlouhou dobu aktualizovány. Obecný problém dokumentací je ten, že je málokdo dělá rád. Proto požadavkem na tento systém byla alespoň minimální schopnost automatické dokumentace.

Automatizace Na produkci je prováděno velké množství úkonů, které by bylo možné tímto systémem automatizovat. Převážně jde o konfigurace apache pro jednotlivé zákazníky, aktualizace důležitých balíčků systému, hromadná instalace nového software, atp...

Jednotnost Každý administrátor do produkce vnesl vlastní specifická řešení. To obvykle není problém, pokud je řešení nějakého problému použito napříč celou produkcí. Pokud by se například způsob nastavení apache výrazně lišilo mezi jednotlivými výpočetními servery, mohlo by zbytečně docházet k omylům.

Ovladatelnost Čas od času se může stát, že systém bude muset ovládat osoba neznalá detailů produkce. V takovém případě je potřeba, aby i neznalý člověk dokázal provést alespoň základní úkony pro vyřešení problému.

Rozšiřitelnost Systém musí být *jednoduše* rozšiřitelný nad rámec práce vykonané při vzniku tohoto dokumentu. Sebelepší řešení, jenž není možné dále rozšiřovat je k ničemu.

Nezávislost Systém nesmí být vázán licencí omezující jeho použití či úpravy.

Tyto podmínky definují základní rámec *aplikace pro správu konfigurací*, jež je hlavním předmětem této práce. Potřeba spravovat konfigurace serverů historicky vznikla v době,

kdy si jedna společnost mohla dovolit vlastnit více než jeden server. Nástup virtualizačních technologií tuto potřebu silně umocnil. Jeden hardwarový server dokáže dnes v jistých situacích hostovat až desítky serverů virtuálních. Jakkoli velká skupina administrátorů v dnešní době jednoduše nemůže ručně spravovat takové množství strojů. Pojďme si tedy představit v dnešní době nejznámější Configuration Management Software (CMS).

3.1 Přehled CMS

3.1.1 cfengine

Cfengine je software pro správu konfigurací založený na modelu klient-server. Kterýkoliv klient, jež se svojí konfigurací odkloní od požadovaného stavu, je do něj zpět uveden. Stav konfigurace je uváděn pomocí deklarativního jazyka. Osobně mi *cfengine* svojí syntaxí přišel velmi matoucí. Při pokusu o sepsání základní konfigurace testovacího serveru nebylo zřejmé „jak na to“. Syntaxe jazyka pro mě nemá nějaký zřejmý, „pohledově“ přehledný formát. Systém sám o sobě nabízí rozumné množství podpůrných prostředků. Instalace balíčků, definice souborů a jejich obsahu, definice mountů, atp... Mezi velké společnosti využívající *cfengine* patří například: AMD, Disney, Nokia, FedEx. Úplný výčet je k nalezení na webu *cfengine* [4]. První oficiální vydání je datováno rokem *1993*, zdrojové kódy jsou v jazyce *C* a licence je GNU General Public License (GPL).

3.1.2 chef

Chef je napsán v ruby a je licencován pod *Apache License*. První vydání je datováno k počátku roku *2009*. Při seznamování s tímto softwarem jsem měl velké problémy s pochopením principu fungování a základních principů. Základní nastavení jednoho serveru a jednoho klienta trvalo velmi dlouho a vyžadovalo velkou řadu kroků. Konfigurační soubory jsou rozšířený jazyk Ruby, což někteří lidé pohybující se v oboru konfiguračního managementu pouvažují za nevýhodu [13]. Za velkou nevýhodu považuji nestandardní postup instalace na námi primárně používané distribuci - Debian.

3.1.3 puppet

Puppet je napsán v Ruby, vydáván je pod *Apache License* a první vydání je datováno k roku 2005. Je obsažen jako standardní balíček ve všech námi používaných distribucích. Syntax konfigurací je velmi jednoduchý, příjemný na čtení a přehledný. Puppet používají například firmy Oracle SUN, digg, twitter, a další... Tento software nabízí nativní podporu pro správu velkého množství různých součástí systému.

3.1.4 Závěr

Výčet CMS zde uvedený rozhodně není nijak vyčerpávající a nezachází do detailů možností a vlastností jednotlivých softwarových produktů. Při mém seznamování s jednotlivými CMS jsem se u každého setkal s problémy, nebo nepříjemnými vlastnostmi, jež mne od použití právě toho či onoho produktu zrazovaly. Nejlépe v mém experimentování dopadl puppet, který byl velmi jednoduchý na instalaci a základní konfiguraci. Nabízí velmi snadné návody pro začátečníky. Důvod pro zvolení tohoto produktu jako našeho systému pro správu konfigurací je tedy kombinací následujících faktorů: časový pres nutící technologie používat namísto toho si s nimi „hrát“, velmi jednoduchá počáteční konfigurace a strmá křivka učení, rozsáhlá dokumentace s kvalitními příklady jak na oficiálním webu, tak na webech fanoušků, pocitově výrazné zaměření dnešní internetové komunity konfiguračních manažerů právě na puppet.

Výběr právě tohoto softwaru není podepřen rozsáhlou analýzou schopností softwaru, jeho budoucího vývoje, či vhodnosti použití pro náš případ. Firma jako taková se nezabývá dodáváním softwaru pro správu konfigurací. Puppet byl původně zamýšlen jako okrajový podpůrný software, ale přirozenou evolucí z něj vznikl mocný nástroj pro podporu serverových procesů firmy. Obhájení mého výběru tedy bude stát na zpětném pohledu a prokázání toho, že nasazení právě této technologie bylo pro firmu přínosem. Samozřejmě pokud bychom během adopce této technologie narazili na závažné problémy, pak by se práce pozastavily a započali bychom s bližším zkoumáním jiné technologie.

3.2 Puppet

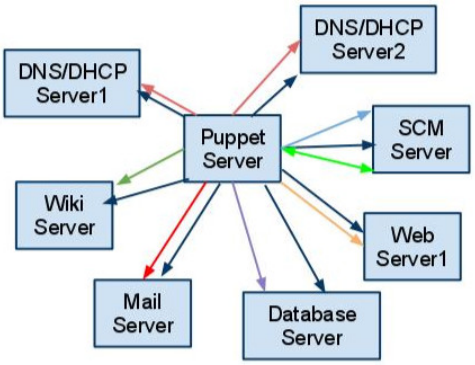
Software puppet byl vybrán pro náš projekt „převzetí produkce“. Pod pojmem převzetí je myšleno zmapování veškerého používaného softwaru, vytvoření produkční dokumentace, sjednocení používaných postupů napříč servery a automatizace často používaných postupů. To je poměrně široký záběr úkolů na jeden software. Pojďme si tedy nejprve probrat základní principy fungování puppetu.

Puppet má velmi specifický jazyk pro popis stavu systému. Vzniklo tedy názvosloví charakteristické pro tento jazyk. Pojmy jsou občas nápadně podobné pojmům ze světa „tradičního“ programování. Význam je obvykle trochu posunutý. Proto bych se nyní rád odkázal na anglický slovník stažený přímo z webu vývojářů puppetu [19]. Slovník lze nalézt v příloze C.1

3.2.1 Jak puppet funguje

Hlavní cíl vývojářů puppetu je poskytnout dostatečně elegantní a efektivní jazyk pro popis konfigurace (stavu) systému. Tento jazyk musí umožnit definovat stav všech prvků operačního systému a následně tyto prvky do oněch definovaných stavů uvést. Puppet

Obrázek 3.1: Model klient-server softwaru puppet



tedy vynucuje stav operačního systému dle konfiguračních souborů, takzvaných *manifestů*. Využívá přitom architektury *klient - server*. Klientem se rozumí stroj, jenž je konfigurován. Serveru se běžně říká *puppet master* a právě na něm je uložena konfigurace všech klientů. Klienti se ve standardním scénáři pravidelně připojují na server a žádají ho o svou konfiguraci. Tato situace je znázorněna na obrázku 3.1. Dotazování je zajištěno spuštěným démonem na každém klientovi. Obdobný démon běží i na serveru a „kompletuje“ konfigurace pro jednotlivé klienty z jednotlivých *manifestů* do takzvaného *katalogu*. Tato situace je vidět na obrázku 3.2.

Instalace puppetu je jednoduchá. V distribuci Debian stačí přes balíčkovací systém nainstalovat balíček *puppet* - na klientských strojích či *puppetmaster* - na serveru. Jelikož serverový stroj sám o sobě také potřebuje konfigurovat, je doporučeným postupem nastavit ho současně i jako klientský stroj. Pro základní běh serveru není potřeba dělat nic speciálního. Klient potřebuje vědět na jakém stroji běží server, viz ukázka ve zdrojovém kódu 3.2. Spojení klientů se serverem je závislé na autentikaci pomocí certifikátů. Pro správné fungování si klient u serveru musí zažádat o registraci svého veřejného klíče. Teprve po registraci tohoto klíče je povoleno obsluhování klienta serverem. Tento postup je vidět ve výpisu konzole 3.1.

```
1 # Zazadame o zaregistrovani certifikatu na serveru
2 web03:~ # puppetd --waitforcert 60 --debug --verbose --no-daemonize 2> /dev/null
3 # Pro zobrazeni prave cekajicich registraci
4 dev01:~ # puppetca --list
5 web03.mydomain.tld
6 # Timto prikazem sparujeme cekajici registraci certifikatu pro dany server
7 dev01:~ # puppetca --sign web03.mydomain.tld
```

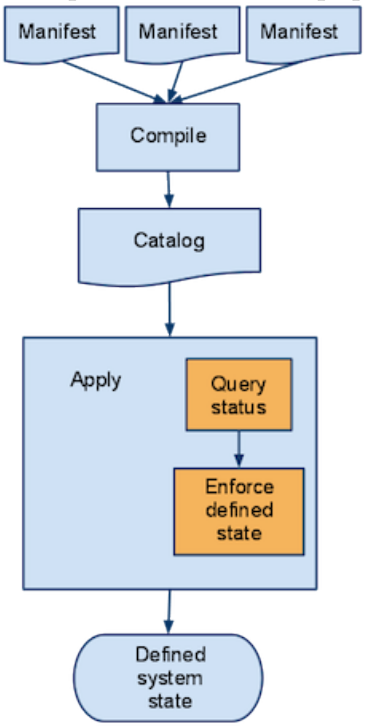
Listing 3.1: Ukázka spárování klienta se serverem - puppet

```
1 [main]
2     server      = dev01.mydomain.tld
```

Listing 3.2: Ukázka konfigurace puppet klienta /etc/puppet/puppet.conf

Po této sérii příkazů by na puppet masteru, jenž je nainstalovaný na stroji *dev01* měl být nainstalován certifikát pro server *web03*. Od této chvíle tedy může server *web03* žádat

Obrázek 3.2: Kompletace manifestů puppet masterem



o jemu patřící katalogy. Velmi častý problém tohoto kroku je ten, že servery nepoužívají plně kvalifikované doménové jméno[1] (Fully Qualified Domain Name (FQDN)), a proto má klient problém s registrací certifikátu na serveru. Doporučeným řešením je používat FQDN na všech místech. Pro záznamy v */etc/hosts*, DNS záznamy, hostname serverů, konfigurace puppetu.

3.2.2 Základy jazyka manifestů puppetu

Puppet master vždy začíná souborem *manifests/site.pp*, který při kompletaci katalogu načte do paměti a hledá záznam typu *node*, jenž se shoduje s hostname právě obsluhovaného klienta. Poté nalezne odkazy na veškeré *zdroje*[2], jejich závislosti, potřebné soubory a vytvoří z nich katalog[3].

Příručka *Puppet best practices 2.0* [14] je dle mého názoru velmi kvalitní referencí k základnímu fungování puppetu a návod jak psát kvalitní puppet manifesty. Tato příručka doporučuje použít soubor *manifests/site.pp* pouze pro definici prvků společných všem serverům či globálních definic a definici jednotlivých serverů vložit do souboru *manifests/nodes.pp*, jenž bude v *site.pp* načítán.

Jelikož nemáme dokonale heterogenní prostředí, umístili jsme do *site.pp* také definice, jež může využívat kterýkoliv ze serverů. Ukázka tohoto souboru je ve zdrojovém kódu B.2. Takovéto definice se velmi hodí při změně verze, či distribuce operačního systému. Na tomto

[1] Například: web03.mydomain.tld

[2] Zdrojem se v puppetu rozumí nějaká vlastnost systému. Například soubor, nastavení nějaké hodnoty OS, existence balíčku, atp...

[3] Postup je reálně daleko složitější.

příkladu je hezky vidět práce puppetu s elementem typu *package*. Puppet používá takzvané providery, neboli poskytovatele pro obsluhu jednotlivých balíčkovacích systémů. Poskytovatelé se mění podle použité distribuce OS klienta. Pokud je na klientu použit Debian, pak se použije provider *apt*, pokud Gentoo, pak bude použit poskytovatel *portage*. Je tak teoreticky jedno, jakou distribuci puppet obsluhuje, přístup ve všech ostatních manifestech by měl být na použité distribuci nezávislý[4].

Pro získávání informací o operačním systému klienta puppet používa open-source Ruby knihovnu *facter*. Facter má za úkol přinášet jednotné rozhraní pro získávání základních ifnormací o serveru bez závislosti na typu OS či specifikách daných distribucí. Ukázka práce facteru je vidět na zdrojovém kódu 3.3

```
1  dev01:~ # facter
2  architecture => amd64
3  domain => mydomain.tld
4  facterversion => 1.5.1
5  fqdn => dev01.mydomain.tld
6  hardwareisa => unknown
7  hardwaremodel => x86_64
8  hostname => dev01
9  id => root
10 interfaces => eth0,tap
```

Listing 3.3: Ukázka práce knihovny facter

Puppet dle dokumentace [17] nativně podporuje velké množství *typů* zdrojů. Je tak možné spravovat záznamy v */etc/crontab*, přidávat systémové balíčky, instalovat soubory konfigurací, definovat pravidla firewallu, přidávat a odebírat systémové uživatele, upravovat systémové mounty, uklízet „nepořádek“ na disku, atp... Možností je opravdu hodně a neexistuje pouze jeden postup pro dosažení nějakého cíle.

Puppet může využívat takzvané *erb* templaty pro definici obsahu souborů. *ERB* template není nic složitějšího než šablona, do níž jsou na zástupná místa doplněny hodnoty proměnných. *ERB* samozřejmě umožňuje větvení, cykly, proměnné a další vlastnosti známé z běžných programovacích jazyků. Vzniká tak elegantní možnost nahrání konfiguračních souborů jednotlivých balíčků OS, aniž by administrátor musel být odkázán na neflexibilní statické soubory. Více o *ERB* si lze přečíst v dokumentaci Ruby [5] či v dokumentaci puppetu [18]. Šablony jsou uloženy v podadresáři */template*.

Jazyk puppetu umožňuje definovat závislosti a to hned dvojího druhu:

require Pokud je instance nějakého zdroje závislá na jiné instanci, lze tuto závislost označit touto vlastností. Puppet poté zařídí, že instance, na níž je naše instance závislá, bude vyhodnocena dříve.

notify Pokud se daná instance změní (nejčastěji změna obsahu souboru), pak je instance, na níž ukazuje tato vlastnost, o této změně uvědoměna. Obvykle se jedná o restart

[4]Samozřejmě situace není tak jednoduchá a určité problémy díky různým distribucím zůstanou.

démona v případě změny jeho konfiguračních souborů.

Puppet umožňuje tvorbu modulů. Modulem se rozumí ucelená množina manifestů, šablon, souborů, providerů, podpůrných skriptů, atp... Princip modulu je takový že při přenesení jeho adresáře na jiné prostředí, je tento modul schopen fungovat. Jedná se tedy o jakousi organizační jednotku ve struktuře puppetu. Uzavřený celek, jenž obstarává nějakou funkcionalitu. Modul může míst závislosti na jiné moduly.

Ve standardní konfiguraci se puppet klient připojuje na server jednou za *30 minut*. Tato doba může být v krizových situacích neakceptovatelně dlouhá, a nejen tehdy. Mohou například existovat případy, kdy je potřeba provést změnu na více serverech najednou. Tehdy by se hodila možnost „nakopnout“ klienty, aby si zažádali o nový, aktualizovaný katalog. Tato možnost opravdu existuje a v puppetu se jí příhodně říká *puppet kick*. Tuto možnost má puppet master, a to pouze tehdy, pokud je k tomu daný klient nakonfigurovaný, a pokud na klientu běží puppet démon. Ukázka skriptu, jenž využívá této vlastnosti, je vidět na zdrojovém kódu 3.4.

```
1  #!/bin/bash
2
3  NODES=(web02 web03 web04 web05)
4  DOMAIN='mydomain.tld'
5
6  for node in "${NODES[@]}"; do
7          puppet kick --foreground --debug "${node}.${DOMAIN}"
8  done
```

Listing 3.4: Ukázka puppet kick

Konfiguraci puppetu máme uloženou v svn. Pro větší pohodlí práce jsme využili schopností svn serveru reagovat při určitých událostech. Vytvořili jsme svn hook, který při každém commitu zkontroluje, zda nebyly změněny soubory puppetu. Pokud ano, automaticky je aktualizuje u puppet mastera. Tento skript je uložen na svn serveru v *hooks/post-commit* a jeho obsah je vidět na zdrojovém kódu 3.5

```
1  #!/bin/sh
2
3  REPOS="$1"
4  REVISION="$2"
5
6  svnlook changed ${REPOS} | grep 'puppet'
7  ret_code=$?
8
9  if [ $ret_code -eq 0 ]; then # Was puppet repo changed?
10     svn up /etc/puppet/ # Update puppet master repository
11 fi
```

Listing 3.5: SVN hook pro automatickou aktualizaci puppet mastera

3.2.3 Případy užití puppetu

Hlavní část práce této kapitoly byla již částečně obsažena v kapitolách předchozích. Například ruku v ruce s přechodem produkce na GlusterFS musely být upraveny definice

mountů jednotlivých serverů. V této kapitole se tedy pokusím probrat jednotlivé případy, které puppet obsluhuje, a s kterými nám výrazně zjednodušil práci.

Sjednocení prostředí

Za relativně krátkou dobu, co pracuji v oboru konfiguračního managementu, musím říci, že málokterá vlastnost serverů dokáže naštvat více, než je nejednotnost prostředí. Různé nastavení PS1[5], rozdílné nastavení doplňování tabulátoru, různé chování editoru vim[6], absence základních balíčků, atp... Všechny tyto body dělají práci se servery komplikovanou a nepříjemnou. Díky použití CMS puppet se nám podařilo tyto nepříjemnosti odstranit. Každý server, jenž je puppetem ovládán, dostane automaticky základní sadu balíčků a nastavení některých programů. Pro příklad:

/etc/motd.tail Úvodní zpráva, jenž se vypíše po přihlášení uživatele. Většina distribucí přidává své velmi dlouhé zprávy o licenčním ujednání, naposledy přihlášených uživatelích či informace o tom, kde sehnat nápovědu. Tyto informace rozhodně nejsou potřeba a při práci jen zbytečně rozptylují.

/etc/aliases Většina programů posílá hlášení na účet uživatele *root*. Tento soubor může definovat na jakou e-mailovou adresu pak tento e-mail dojde.

vim Vim je velice mocný textový editor. Osobně neznám lepší nástroj pro úpravu konfiguračních souborů či jen zdrojových kódů. Přes velmi těžké počáteční zaučení vim vřele doporučuji každému administrátorovi.

logwatch Logwatch sleduje logy systému a vytváří pravidelné reporty. Nejednou jsme byli varování na blížící se problém.

puppet Puppet je zde jmenován z důvodů aktualizace tohoto balíčku. Z principu již nainstalovaný být musí.

git Git je distribuovaný verzovací systém. Každý server v mé správě má verzovaný adresář */etc/*. *Velmi* vřele tuto praktiku doporučuji každému, kdo přebírá již existující server po jiném administrátorovi. Omyly se stávají a mít zálohu konfiguračních souborů je často k nezaplacení.

bash Běžně používaný shell. Opět uveden pouze z důvodů pravidelné aktualizace. Současně s puppet modulem pro bash přijde i jednotné nastavení PS1.

snmpd Velké množství služeb a programů souvisejících se správou serverů využívá protokol SNMP. Tento balíček je nutný už z důvodu monitorování probraného v kapitole 2.2

[5]Řádek, jenž je vytištěn do příkazové řádky po každém ukončeném příkazu, jenž předá volání zpět do shellu.

[6]Nebo jiného editoru.

ntp Velké množství služeb a programů pro své správné fungování vyžaduje správně seřízené systémové hodiny. Program NTP se o toto dokáže postarat.

Toto je základní sada balíčků a konfiguračních souborů, jenž považuji za nepostradatelnou na *každém* serveru. Instalaci balíčků jsem v puppetu řešil jako samostatné moduly. Kód se lépe udržuje, zejména pokud k balíčkům patří i nějaké konfigurační soubory či šablony.

Provádět vložení (include) těchto modulů v každém *node* by bylo velmi nepraktické, proto jsem nejprve předpokládal, že vytvořím obecný *node* a od něho poté oddědím všechny ostatní nody. Bohužel v puppetu dědičnost nodů nefunguje jako v běžném objektovém jazyce. Tento fakt zachycuje i dokument *Common puppet misconceptions* [16]. Proměnná, nastavená v potomkovi, se nevypropaguje do rodiče. Třídy vložené v rodiči nelze žádným způsobem z potomka parametrizovat, jednalo by se o značné omezení. Místo tohoto postupu se běžně místo rodičovského node vytváří třída, která se pouze vkládá. Zde předefinování proměnných již funguje. Námi vytvořená třída se jmenuje *base_node_class* a ukázka jejího použití je vidět na zdrojovém kódu 3.6. Tímto jednoduchým kódem si připravíme server *db01* tak, aby se pro nás v základních bodech choval stejně jako všechny ostatní.

```
1 node db01 {
2     include base_node_class
3 }
```

Listing 3.6: Ukázka použití base_node_class

GlusterFS a adresáře uživatelských dat

Jak již bylo probráno v kapitole 1.6.3, server s adresáři uživatelských dat prošel za dobu zpracování tohoto projektu významnými změnami. Především se jedná o přechod z technologie *NFS* na technologii *GlusterFS*. Během této realizace nám puppet významně pomohl s vypropagováním změn na všechny potřebné servery. V původním řešení (NFS) byla uživatelská data řešena modulem *nfs-mounts*, jenž je přiložen k této práci.

Nové řešení, založené na technologii *GlusterFS*, je obsaženo v modulu *gluster*. Tato technologie obsahuje mezi jednotlivými verzemi *razantní* změny. Potřebovali jsme dosáhnout jednotnosti verzí technologie *Gluster* na jednotlivých serverech, bez ohledu na distribuci či verzi distribuce. Toho lze za pomocí puppetu poměrně elegantně dosáhnout. Manifesty napíšeme tak, že si nejprve stáhneme ze serveru balíčky pro jednotlivé distribuce. Poté je, obvykle za pomocí jiných providerů, nainstalujeme. Vše musí být pro správnou funkčnost podpořeno odpovídajícími závislostmi (viz obrázek A.12). Pokud není stáhnut instalační soubor, nemá cenu se pokoušet ho instalovat. Dokud není nainstalován, nemá smysl pokoušet se o připojení adresářů. Stejně tak není možné připojit vzdálené adresáře do doby, než jsou vytvořeny všechny odpovídající lokální adresáře. Atp...

Bootstrap konfigurace serveru

Puppet vlastně nabízí kompletní možnost konfigurace serveru. Ve chvíli, kdy je server v bootovatelném stavu, je nainstalován balík puppet a na puppet masteru je nainstalován certifikát daného stroje. Nyní je možné veškerý životní běh konfigurace serveru řídit přes CMS a zkonfigurovat si tak server přesně podle daných pravidel bez nutnosti ručního zásahu. Tuto teorii jsme si ověřili u serveru *watch01*, jenž byl vybrán pro pokusnou aktualizaci distribuce OS[7]. Bohužel díky mé neznalosti technologie *EC2* se tento projekt nepovedl. Bylo tedy rozhodnuto, že *watch01* bude nainstalován znovu, a to přímo na novější verzi OS. Po instalaci zabere nastavení puppetu přibližně *5 minut*. Poté stačí tento software spustit a za cca *20 minut* je server připravený *přesně* v takové podobě, v jaké byl před započetím výše zmiňovaného projektu. Z této zkušenosti jsme vyvodili následující důsledky:

- Obrovský nárůst možností expanze produkce z pohledu instalace nových serverů. Administrátor má díky CMS jistotu, že nové stroje se budou chovat očekávaným způsobem.
- Díky puppetu máme menší závislost na poskytovateli server-housingu.
- Použití CMS ve výsledku ušetří mnoho času administrátora.

Úklid serveru

Disk každého serveru je dříve či později zaneřáděn velkým množstvím souborů, které byly administrátorem či jinými uživateli vytvořeny a zapomenuty. Časem si žádný z uživatelů nevzpomene na důvod existence toho či onoho souboru. Disky nemají nekonečnou kapacitu, může se stát, že se dříve či později zaplní. Proto vznikla potřeba vytvořit systém, jenž by nepotřebné soubory čas od času promazal.

Puppet nativně zná *typ* tidy. Tento, jak už název napovídá, má za úkol odstraňování nepotřebných souborů. Seznam takových souborů se musí vytvořit časem podle kritérií specifických pro daný server. Ve zdrojovém kódu 3.7 je vidět příklad odstranění běžného nepořádku z */tmp*.

```
1 tidy { "/tmp":
2     recurse => 1, # No subdirectories!
3     backup  => false, # Do not create backups
4     # Usual garbage files pattern
5     matches => [ "*.gz", "*.bz2", "*.gif", "*.sql", "*.log", "*.ico", "*.inc", "*.xsd", "
          *.conf", "*.txt", "*.zip" ],
6     age     => "1m" # Only delete files older than one month
7 }
```

Listing 3.7: Ukázka typu tidy

[7] Z Debian Lenny na Debian Squeeze

Failover lb01

Na obrázku A.7 je nevybarvený server *lb02*. Vznikl pouze náhodou při projektu povýšení distribuce na výpočetních serverech *web02* a *web03*. Bylo zjištěno, že *dom0 lenny*, nemůže hostovat *domU squeeze*. Proto bylo nutné povýšit i *web02-host* a *web03-host*. Na *web02-host* ale běží server *lb01*, který je nezbytný pro běh produkce jako celku. Proto nebylo možné riskovat aktualizaci hardwarového stroje, která v případě problémů mohla ohrozit dostupnost produkce.

Proto bylo rozhodnuto, že po dobu aktualizace *web02-host* bude *lb01* přesunut na *web03-host*. Živá migrace bohužel nebyla možná. Naštěstí ale nemá běh serveru *lb01* vliv na jeho stav. Jinými slovy kopie *lb01*, jenž by nějakou dobu neběžela dokáže plně nahradit funkci původního serveru, jež celou dobu běžel. Pro porovnání: Například databázové stroje, či file-servery takovouto vlastnost nemají. Kopie *lb01* tedy byla vytvořena pomocí Logical Volume Manager (LVM) snapshotu. Vytvoření snapshotu je důležité, protože jinak by během původního stroje mohlo dojít k poškození souborů na stroji novém. Obraz disku byl na druhý hostitelský systém přenesen pomocí programu *scp* a *dd*. V jednu chvíli byly servery ve své funkci vyměněny. Celková doba výpadku nepřesáhla jednu minutu. Praktická ukázka tohoto přenesení je vidět na zdrojovém kódu 3.8.

```
1  web02-host:~ # scp /etc/xen/lb01 web03-host:/etc/xen/
2  web03-host:~ # lvcreate -L 10G -n lb01_root system
3  web02-host:~ # lvcreate -L 10G -s -n lb01_root_backup /dev/system/lb01_root
4  web02-host:~ # dd if=/dev/system/lb01_root_backup bs=4M | pv -s 10g | ssh web03-host dd
       of=/dev/system/lb01_root bs=4M
5  2560+0 records out
6  10737418240 bytes (11 GB) copied, 494.311 s, 21.7 MB/s
7    10GB 0:08:14 [20.7MB/s] [=================================================>] 100%
8  0+557755 records in
9  0+557755 records out
10 10737418240 bytes (11 GB) copied, 494.447 s, 21.7 MB/s
11 # The process took us 8min 12s
12 web02-host:~ # xm shutdown lb01
13 web03-host:~ # xm create lb01
```

Listing 3.8: Ukázka manuální migrace domU pomocí scp a dd.

Tato zkušenost nám vnukla nápad záložního load-balanceru. Ideální řešení by bylo v podobě horké zálohy, například na základě běžícího stroje a protokolu Common Address Redundancy Protocol (CARP). Tato myšlenka ale byla mimo rozsah této práce. A proto jsme pouze zadokumentovali, že v případě selhání *web03-host*[8] stačí zapnout *lb01* na *web02-host* a produkce by měla jen s malým přerušením běžet dále.

V tomto hraje velmi důležitou roli puppet, jenž dokáže po zapnutí náhradního load-balanceru automaticky aktualizovat jeho konfigurace. Bez zapojení CMS do procesu konfigurace produkce by takováto možnost vůbec nevznikla.

[8] Nový domU pro *lb01*

Distribuce pravidel firewallu

Puppet se na produkci stará o distribuci pravidel firewallů na jednotlivé servery. Jde o soubor obsahující definice, jež dokáže přečíst program *iptables-restore*. S komplexními softwary typu *firewall builder* máme špatné zkušenosti. Jejich integrace do systému je poměrně agresivní a administrátor má velkou šanci, že neopatrným počínáním v gui aplikace něco zničí. Proto byl zvolen tento jednoduchý, ale funkční princip.

Výpočetní servery a nastavení virtual hostů

Největší a nejvíce používanou službou, kterou puppet na naší produkci obsluhuje jsou konfigurace *apache* jednotlivých výpočetních serverů. Jedná se především o definice virtual hostů, které se mění relativně často. Modul puppetu nese označení *web-xx*. Sekundární, měně intenzivně využívanou součástí tohoto modulu jsou obecné definice apache a PHP výpočetních serverů.

Modul byl navržen pro maximální jednoduchost při přidávání nových virtual hostů. V adresáři *web-xx/files/apache2/sites-available* jsou umístěny jednotlivé soubory virtual hostů. Tyto jsou automaticky přeneseny na všechny výpočetní servery do */etc/apache2/sites-available*. Apache je po vzoru distribuce debian nakonfigurován, aby automaticky načetl všechny konfigurace z adresáře */etc/apache2/sites-enabled*. Nikoliv tedy z adresáře, do kterého jsou soubory puppetem umístěny. Jejich aktivace je provedena použitím *definovaného typu*[9]. Jinými slovy po syntakticky správném umístění názvu virtual hostu do *web-xx/manifests/vhosts.pp* bude daný virtual host aktivován.

Abych „nevynalézal znovu kolo“, rozhodl jsem se pro modul obsluhující *základní* běh apache použít již napsanou knihovnu puppetu. Nejvíce vyhovující implementaci[10] jsem našel od člověka jménem *Paul Lathrop*, který se v současné době živí jako konfigurační manažer[11]. Princip jeho modulu spočívá v odstranění všech specifik, které s sebou jednotlivé operační systémy přináší a následné konfigurace po vzoru distribuce Debian.

Modul *web-xx* dále jako závislosti přináší modul *gluster* pro adresáře uživatelských dat. Doplňkové balíčky a moduly apache a PHP a jejich konfigurace. Modul *postfix* pro nastavení odesílání pošty výpočetních serverů. Definiční soubor *worker_cookie*, který je nezbytný pro správnou práci load balanceru.

Jednotlivé soubory virtual hostů jsou obyčejné, ručně psané definice konfigurací apache. Zde by se samozřejmě nabízela možnost použití *erb* šablon a dynamického generování virtual hostů, za využití společných částí. Tato možnost se nám nezdála příliš vhodná. Zkompilované virtual hosty by se v případě krizové situace špatně hromadně upravovaly. Navíc zanesení komplexní technologie jakou jsou *erb* šablony, do tak citlivých konfigurací

[9] Viz „defined type“ v slovníku puppetu
[10] `https://github.com/plathrop/puppet-module-apache`
[11] `http://plathrop.tertiusfamily.net/`

jakými jsou virtual hosty, by mohlo mít za následek skryté, těžko odhalitelné chyby. Proto jsme se rozhodli pro použití technologie *mod_macro* popsané v kapitole 2.8.

Kapitola 4

Zhodnocení

Implementace všech projektů popisovaných v této práci probíhala iterativně, na základě aktuálních potřeb. Výstupy a zkušenosti jednoho projektu se obvykle použily v některém z dalších projektů. Některé projekty, jenž dle mého soudu jen okrajově souvisely s tématem konfiguračního managementu a správy serverů jsem vynechal úplně. Získané zkušenosti z těchto projektů jako celku bych shrnul do následujících bodů:

- Kvalitní monitorovací systém je základem každého produkčního prostředí. Pokud se objevuje jen nepatrné procento falešně pozitivních hlášení, administrátoři začnou tato hlášení velice rychle ignorovat a snadno se stane, že přehlédnou nějaké opodstatněné hlášení. Současně je potřeba zajistit, aby monitorovací služba sledovala všechny známé problémové části systému.

- Osobní znalost chování produkčního prostředí je nenahraditelná a nemůže jí zastoupit sebelepší dokumentace. Je proto dobré pravidelně distribuovat informace ohledně změn produkčního prostředí mezi více lidí. A zavést zvyk, že opakující se problémy nespravuje pouze jeden člověk.

- Dokumentace by nikdy neměla být podceňována. I pouhý výpis z konzole přiložený k tiketu je lepší, než žádná dokumentace.

- Je dobré provést řešení daného problému v duchu Keep It Simple and Stupid (KISS) principu. Do složitých (komplexních) řešení je těžší proniknout. Jsou obvykle hůře modifikovatelná. A z mé osobní zkušenosti přináší jen malou přidanou hodnotu. Je dobré se držet pravidla *80/20*[1].

- Žádné navýšení výkonu HW nemůže vyřešit výkonnostní problémy špatně napsaného Software (SW).

- Don't Repeat Yourself (DRY) princip je velice často zanedbávaný, což vede obvykle k aplikování postupu *Ctrl+C, Ctrl+V*. Následně je kód či konfigurace velmi těžko

[1] Pravidlo 80/20 říká, že 80 procent všech výsledků vzniká z 20 procent příčin.

udržovatelný. Kopírování konfigurací se zprvu může zdát jako nejrychlejší řešení, ale ve výsledku obvykle zabere daleko více času než dodržování DRY přístupu.

- CMS puppet nám, přes velmi vysoké počáteční náklady, ušetřil velké množství času při správě serverů.
- Puppet je jednoduchý nástroj pro tvorbu dokumentace konfigurací produkčních serverů. Tím, že jsou konfigurace puppetu v svn, můžeme jednoduše sledovat motivace jednotlivých záznamů v konfiguracích. Jako dokumentace slouží i komentáře v jednotlivých manifestech puppetu.
- CMS velmi zpříjemňuje každodenní interakci se servery, jelikož dokáže sjednotit přístup administrátora k různým OS či verzím OS.
- S větší integrací puppetu do naší produkce, přímo úměrně vzrůstá míra zásahů, jenž v systému dokážeme provést za jednotku čacu.
- Před implementací jakékoliv změny je dobré z dostupných dat vytvořit hypotézy ohledně výkonnostních dopadů, sepsat seznam systémů na které budou mít změny vliv a určit riziková místa implementace. Je možné, že se ukáže, že změna nepřinesla kýžený výkonnostní efekt. Nebo že změna ovlivní systém takovým způsobem, jenž si nemůžeme dovolit.
- Murphyho zákon: „Co se může pokazit, to se pokazí. Co se nemůže pokazit, to se pokazí taky, ale až za delší dobu“ je platný.
- Tvrzení: „Každý zdrojový kód o alespoň dvou řádcích obsahuje alespoň jednu chybu.“ je také platné.

4.1 Závěr

V tomto dokumentu jsem se snažil popsat mé seznámení s produkcí firmy, mojí motivaci pro implementaci CMS puppet a jednotlivé projekty, jenž byly do této doby udělány. Provádění změn na produkci, bylo díky brzké integraci puppetu velmi zjednodušeno. Projekty v tomto dokumentu popsané trvaly bezmála jeden kalendářní rok. Z toho přibližné 6 měsíců je využíván puppet. Některé projekty dopadly neúspěchem a jediný přínos měly v podobě poučení pro projekty budoucí. Produkce není v žádném případě v dokonalém stavu. Proces vylepšování stále pokračuje. Z projektu vzešlo velké množství zkušeností a dalších dílčích projektu, které čekají na realizaci.

Slovník

"MySQL" MySQL je databázový systém, vytvořený švédskou firmou MySQL AB, nyní vlastněný společností Sun Microsystems, dceřinou společností Oracle Corporation. Jeho hlavními autory jsou Michael „Monty“ Widenius a David Axmark. Je považován za úspěšného průkopníka dvojího licencování – je k dispozici jak pod bezplatnou licencí GPL, tak pod komerční placenou licencí.. 18, 30, 37

"brute force" Útok hrubou silou (anglicky brute force attack) je většinou pokus o rozluštění šifry bez znalosti jejího klíče k dešifrování. V praxi se jedná o systematické testování všech možných kombinací nebo omezené podmnožiny všech kombinací.. 19

"nagios" Nagios je populární open source systém pro automatizované sledování stavu počítačových sítí a jimi poskytovaných služeb. Je vyvíjen primárně pro Linux, ale je možné ho provozovat i na jiných unixových systémech. Je vydáván pod GPL licencí. Je vyvíjen a udržován Ethanem Galstadtem a mnoha dalšími vývojáři pluginů.. 22, 24

"scale out" Škálování do šířky je obvykle řešeno přidáním serverů a rozdistribuováním zátěže mezi tyto jednotlivé servery. 13

"scale up" Škálování do výšky. Škálování při kterém jsou do serveru přidány výpočetní zdroje (CPU, RAM) pro zvýšení výkonu. 13

ad-hoc Ad hoc [ad hók] (někdy Ad-hoc) je latinský termín, znamenající doslova „k tomuto“, překládaný jako „za určitým účelem“ nebo „pro tento jednotlivý (konkrétní) případ“.. 43

AJAX Asynchronous JavaScript and XML. 39

apache Apache HTTP Server je softwarový webový server s otevřeným kódem pro GNU/Linux, BSD, Solaris, Mac OS X, Microsoft Windows a další platformy. V současné době dodává prohlížečům na celém světě většinu internetových stránek.. 32, 41–43

API Application Programming Interface. 29, 38

ARP Address Resolution Protocol (ARP) se v počítačových sítích s IP protokolem používá k získání ethernetové MAC adresy sousedního stroje z jeho IP adresy. Používá se v situaci, kdy je třeba odeslat IP datagram na adresu ležící ve stejné podsíti jako odesilatel. Data se tedy mají poslat přímo adresátovi, u něhož však odesilatel zná pouze IP adresu. Pro odeslání prostřednictvím např. Ethernetu ale potřebuje znát cílovou ethernetovou adresu.. 23

AWS Amazon Web Services. 28, 29

CARP Common Address Redundancy Protocol. 53

cli Příkazový řádek (zkratka CLI, anglicky Command Line Interface) představuje uživatelské rozhraní, ve kterém uživatel s programy nebo operačním systémem komunikuje zapisováním příkazů do příkazového řádku. Na rozdíl od textového rozhraní a grafického uživatelského rozhraní nevyužívá myš ani menu a nedovede pracovat s celou plochou obrazovky (terminálu).. 24, 29

cloud Cloud computing je na Internetu založený model vývoje a používaní počítačových technologií. Lze ho také charakterizovat jako poskytování služeb či programů uložených na serverech na Internetu s tím, že uživatelé k nim mohou přistupovat například pomocí webového prohlížeče nebo klienta dané aplikace a používat prakticky odkudkoliv. Uživatelé neplatí (za předpokladu, že je služba placená) za vlastní software, ale za jeho užití. Nabídka aplikací se pohybuje od kancelářských aplikací, přes systémy pro distribuované výpočty, až po operační systémy provozované v prohlížečích, jako je například eyeOS, Cloud či iCloud.. 25

CMS Configuration Management Software. 44, 45, 50, 52, 53, 57

CPU Central Processing Unit. 28

cron Cron je softwarový démon, který v operačních systémech automatizovaně spouští v určitý čas nějaký příkaz resp. proces (skript, program apod.). Jedná se vlastně o specializovaný systémový proces, který v operačním systému slouží jakožto plánovač úloh, jenž umožňuje opakované spouštění periodicky se opakujících procesů (např. noční běhy dávkových úloh při hromadném zpracování dat) apod.. 24

Debian Debian GNU/Linux je jednou z nejstarších doposud vyvíjených distribucí GNU/Linuxu, kterou nevyvíjí komerční subjekt, ale je připravována velkým množstvím dobrovolníků z celého světa. Je známa především svou konzervativností. Přesto je to jedna z nejrozšířenějších linuxových distribucí na světě.. 44, 46, 48, 54

DNS Domain Name Server. 21, 24, 37, 47

dom0 Dom0, nebo *domain zero* je první server jenž je spuštěn hypervizorem XENu při bootu. 9, 11

domU DomU, nebo *domain U* je každý další server jenž je spuštěn „nad“ dom0. 8, 11, 12

DRY Don't Repeat Yourself. 56, 57

EC2 Elastic Compute Cloud. 28, 29

failover Schopnost prostředí bezchybného běhu i za situace selhání některých serverů. 1, 13, 18, 19, 34, 36, 37

FQDN Fully Qualified Domain Name. 47

FUSE Filesystem in userspace. 34

GlusterFS GlusterFS je open source, clusterový souborový systém, schopný pojmout data v řádech petabajtů a obsluhovat tisíce klientů. Servery zapojené do GlusterFS tvoří stavební bloky přes Infiniband RDMA a/nebo TCP/IP propojení. Sdílí přitom diskový prostor, operační paměť pro správu dat v jediném globálním jmenném prostoru. GlusterFS je založeno na modulárním designu v uživatelském prostoru a dokáže přinést výjimečný výkon pro různé druhy úloh.. 49

GPL GNU General Public License. 44

HN Hardware Node. 9–11

hostname Hostname je jméno identifikující daný server. Používá se v systému DNS, kdy IP adrese může být přidělen hostname. Častější je obrácený případ, kdy známe hostname a DNS serveru se ptáme na korespondující IP adresu. V tomto případě jde zejména o zjednodušení, aby si člověk nemusel pamatovat IP adresy. V případě že hostname je ve formátu host.mydomain.tld, pak hovoříme o takzvaném plně kvalifikovaném doménovém jménu, neboli FQDN.. 47

HTTP Hypertext transfer protocol je dnes nejrozšířenější protokol pro distribuci obsahu webových stránek. 3, 12, 13, 22, 30

HTTPS Hypertext transfer protocol secure je nadstavba nad klasický HTTP protokol o zabezpečení v podobě protokolu *SSL/TLS*. 13, 22

HW Hardware. 13, 18, 56

IM Instant messaging. 39

IP Internet Protocol. 22–24

KISS Keep It Simple and Stupid. 56

kontejner Kontejner, anglicky *container* je pojem známý především z prostředí *OpenVZ*. 8

KVM Kernel-based Virtual Machine. 10

Load balancer Server jež zajišťuje rovnoměrné rozmístění zátěže na výpočetních strojích. 3

LVM Logical Volume Manager. 53

memcached Kešovací server, jenž ukládá data do paměti. Vyznačuje se rychlým přístupem a velmi omezenou množinou funkcí.. 38, 39

Mooreův zákon Složitost součástek se každý rok zdvojnásobí při zachování stejné ceny `http://en.wikipedia.org/wiki/Moore%27s_law`. 9

mount „Mountování“, je v IT pojem pro označení procesu připravení diskového oddílu pro použití operačním systémem. 13

NFS Network File Storage - protokol pro připojování síťových disků. 3, 13, 17, 30, 31, 33, 34, 36, 38, 39, 51

NSCA Nagios Service Check Acceptor. 27

NTP Network Time Protocol. 26, 51

OID Object Identifier. 24

open source Software s volně dostupnými zdrojovými kódy. 10

OS Operační systém. 3, 5–12, 21, 48, 52, 57

OSS Open Source Software. 19

overhead Zdroje jež musí být vydány navíc a nesouvisí přímo s požadovaným cílem. 10

packet loss Packet loss je jev při kterém jeden, nebo více paketů v počítačové síti nedosáhne svého určeného cíle.. 22

PHP Hypertext Preprocessor. 38, 39, 54

POSIX POSIX (zkratka z Portable Operating System Interface) je přenositelné rozhraní pro operační systémy, standardizované jako IEEE 1003 a ISO/IEC 9945. Vychází ze systémů UNIX, a určuje, jak mají POSIX-konformní systémy vypadat, co mají umět, co se jak dělá apod.. 34

PS1 Řádek, jenž je vytištěn do příkazové řádky po každém ukončeném příkazu, jenž předá volání zpět do shellu.. 50

puppet Puppet je open-source nástroj pro správu konfigurací. Je napsán v Ruby a je vydáván pod licencí GPL až do verze 2.7.0. Od této verze dále je vydáván pod licencí *Apache 2.0*.. 44, 45, 47–54, 57

Python Python je dynamický objektově orientovaný skriptovací programovací jazyk, který v roce 1991 navrhl Guido van Rossum. Python je vyvíjen jako open source projekt, který zdarma nabízí instalační balíky pro většinu běžných platforem (Unix, Windows, Mac OS); ve většině distribucí systému Linux je Python součástí základní instalace.. 23

RAID Redundant Array of Independent Disks. 14, 15

Rolling updates Aktualizace se uskutečňuje pomocí balíčkovacího systému průběžně, denně jsou do zdrojů doplňovány nejnovější stabilní verze softwaru. 7

rsync rsync je počítačový program původně pro Unixové systémy, který synchronizuje soubory a adresáře mezi různými místy za použití co nejmenšího přenosu dat. Tam, kde je to možné, přenáší pouze rozdíly mezi soubory (delta).. 35

RTT Round Trip Time. 30, 31

Ruby Ruby je interpretovaný skriptovací programovací jazyk. Díky své jednoduché syntaxi je poměrně snadný k naučení, přesto však dostatečně výkonný, aby dokázal konkurovat známějším jazykům jako je Python a Perl. Je plně objektově orientovaný – vše v Ruby je objekt.. 44, 48

SNI Server Name Indication. 22

SNMP Simple Network Management Protocol. 23, 24, 26, 50

SPOF Single Point Of Failure. 16, 17

SQL Structured Query Language je standardizovaný dotazovací jazyk používaný pro práci s daty v relačních databázích. 3

SSH Secure Shell. 26

SSL Secure Sockets Layer. 22, 23

svn subversion. 39, 49, 57

SW Software. 56

trac Trac je open source webový systém pro správu projektů a tiketů. Program je inspirován softwarem CVSTrac a byl původně pojmenován svntrac kvůli své schopnosti napojení na Subversion. Je vyvíjen a udržován firmou Edgewall Software.. 24

vim Vim je open source textový editor, který lze spustit v prostředí většiny operačních systémů. Je oblíbený zejména mezi zkušenými uživateli operačních systémů unixového typu. Kromě klasického Vimu existuje celá řada editorů, které jsou založeny na principu Vimu, ale mají nějaké specifické vlastnosti např. KVim pro prostředí KDE.. 50

VPC Virtual Private Cloud. 28–30

VPN Virtual Private Network. 3, 29

XEN V IT se pod pojmem XEN rozumí virtualizační technologie. 11, 12

Literatura

[1] Amazon ec2 - api reference. http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-api.pdf, 2011. [Online; přístupné 10.04.2011].

[2] Amazon ec2 - command line reference. http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-clt.pdf, 2011. [Online; přístupné 10.04.2011].

[3] Amazon elastic compute cloud - user guide. http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-ug.pdf, 2011. [Online; přístupné 10.04.2011].

[4] Cfengine AS. cfengine users. http://www.cfengine.com/pages/companies, 2011. [Online; přístupné 12.04.2011].

[5] James Britt and Neurogam. Erb — ruby templating. http://ruby-doc.org/stdlib/libdoc/erb/rdoc/classes/ERB.html, 2011. [Online; přístupné 17.04.2011].

[6] Fabien Coelho. Apache 2.2 module mod_macro 1.1.11. http://www.cri.ensmp.fr/~coelho/mod_macro/, 2010. [Online; přístupné 20.04.2011].

[7] GlusterFS community. Gluster 3.1: Understanding replication. http://www.gluster.com/community/documentation/index.php/Gluster_3.1:_Understanding_Replication, 2010. [Online; přístupné 11.04.2011].

[8] GlusterFS community. Glusterfs 3.1 documentation. http://gluster.com/community/documentation/index.php/Gluster_3.1:_Introducing_GlusterFS, 2010. [Online; přístupné 11.04.2011].

[9] Laura DiDio. Yankee Group 2007-2008 Server OS Reliability Survey. http://www.iaps.com/exc/yankee-group-2007-2008-server-reliability.pdf, 2008. [Online; přístupné 12.12.2010].

[10] Dormando. What is memcached? http://memcached.org/, 2009. [Online; přístupné 22.05.2011].

[11] edpin,wolf,luiz@google.com. Failure trends in a large disk drive population. http://labs.google.com/papers/disk_failures.html, 2007. [Online; přístupné 22.3.2011].

[12] The Apache Software Foundation. Ssl/tls strong encryption: Faq. l, 2011. [Online; přístupné 29.3.2011].

[13] John from Bitfield Consulting. Puppet versus chef: 10 reasons why puppet wins. http://bitfieldconsulting.com/puppet-vs-chef, 2010. [Online; přístupné 13.04.2011].

[14] Digant C Kasundra. Puppet best practices 2.0. http://projects.puppetlabs.com/projects/puppet/wiki/Puppet_Best_Practice, 2010. [Online; přístupné 15.04.2011].

[15] Kirill Kolyshkin. Virtualization in Linux. http://download.openvz.org/doc/openvz-intro.pdf, 2006. [Online; přístupné 9.2.2011].

[16] Puppet Labs. Docs: Troubleshooting. http://docs.puppetlabs.com/guides/troubleshooting.html#common-misconceptions, 2011. [Online; přístupné 19.04.2011].

[17] Puppet labs. Docs: Type reference. http://docs.puppetlabs.com/references/latest/type.html, 2011. [Online; přístupné 16.04.2011].

[18] Puppet Labs. Docs: Using puppet templates. http://docs.puppetlabs.com/guides/templating.html, 2011. [Online; přístupné 17.04.2011].

[19] Puppet Labs. Glossary of terms. http://projects.puppetlabs.com/projects/puppet/wiki/Glossary_Of_Terms, 2011. [Online; přístupné 17.04.2011].

[20] linux.com. Benchmarking hardware raid vs. linux kernel software raid. http://www.linux.com/news/hardware/servers/8222-benchmarking-hardware-raid-vs-linux-kernel-software-raid, 2008. [Online; přístupné 20.3.2011].

[21] Microsoft. Compare Windows to Red Hat. http://www.microsoft.com/windowsserver/compare/windows-server-vs-red-hat-linux.mspx, 2003. [Online; přístupné 13.12.2010].

[22] VMware, Inc. A Performance Comparison of Hypervisors . http://www.cc.iitd.ernet.in/misc/cloud/hypervisor_performance.pdf, 2007. [Online; přístupné 10.1.2011].

[23] Wikipedia. Failover. http://en.wikipedia.org/wiki/Failover, 2011. [Online; přístupné 22.2.2011].

[24] Wikipedia. Gaussova funkce. http://cs.wikipedia.org/wiki/Gaussova_funkce, 2011. [Online; přístupné 11.04.2011].

[25] Wikipedia. Kernel-based Virtual Machine. http://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine, 2011. [Online; přístupné 2.2.2011].

[26] Wikipedia. Network file system. http://cs.wikipedia.org/wiki/Network_File_System, 2011. [Online; přístupné 27.3.2011].

[27] Wikipedia. RAID. http://cs.wikipedia.org/wiki/RAID#RAID_1_.28zrcadlen.C3.AD.29, 2011. [Online; přístupné 20.3.2011].

[28] Wikipedia. Souběh. http://cs.wikipedia.org/wiki/Soub%C4%9Bh, 2011. [Online; přístupné 05.04.2011].

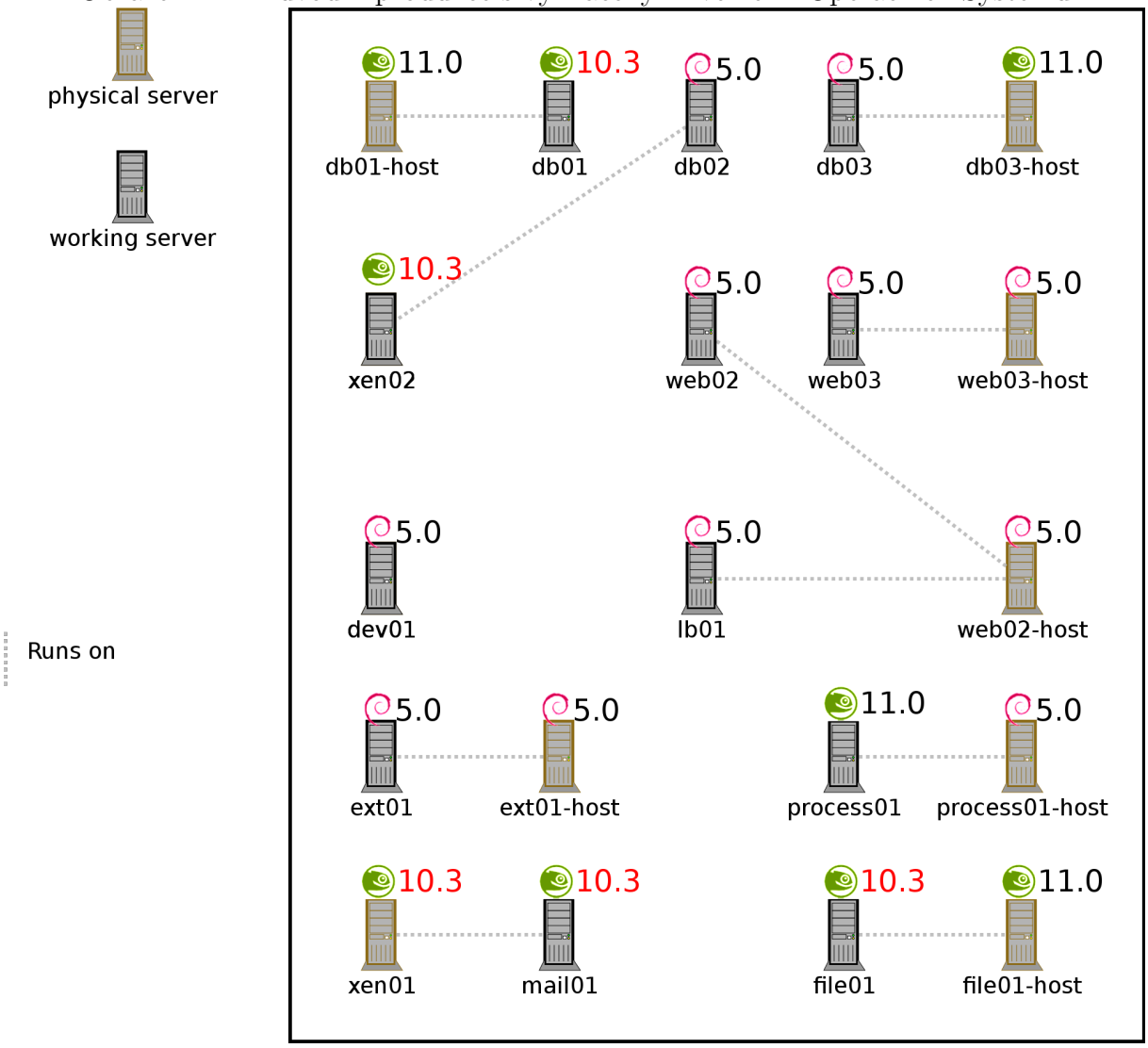
[29] Wikipedie. Round-trip delay time. http://en.wikipedia.org/wiki/Round-trip_delay_time, 2011. [Online; přístupné 11.04.2011].

[30] Cybersource XXX. n. http://www.cyber.com.au/about/linux_vs_windows_tco_comparison.pdf&ei=SyYVTcPQPMWa8QOK_vTfDw&usg=AFQjCNGNPQMsfTKO_AU5gBC6gIOsjGxxIA&sig2=f8P3b0FOgaM-DnCdQq_eRQ, 2002. [Online; přístupné 13.12.2010].

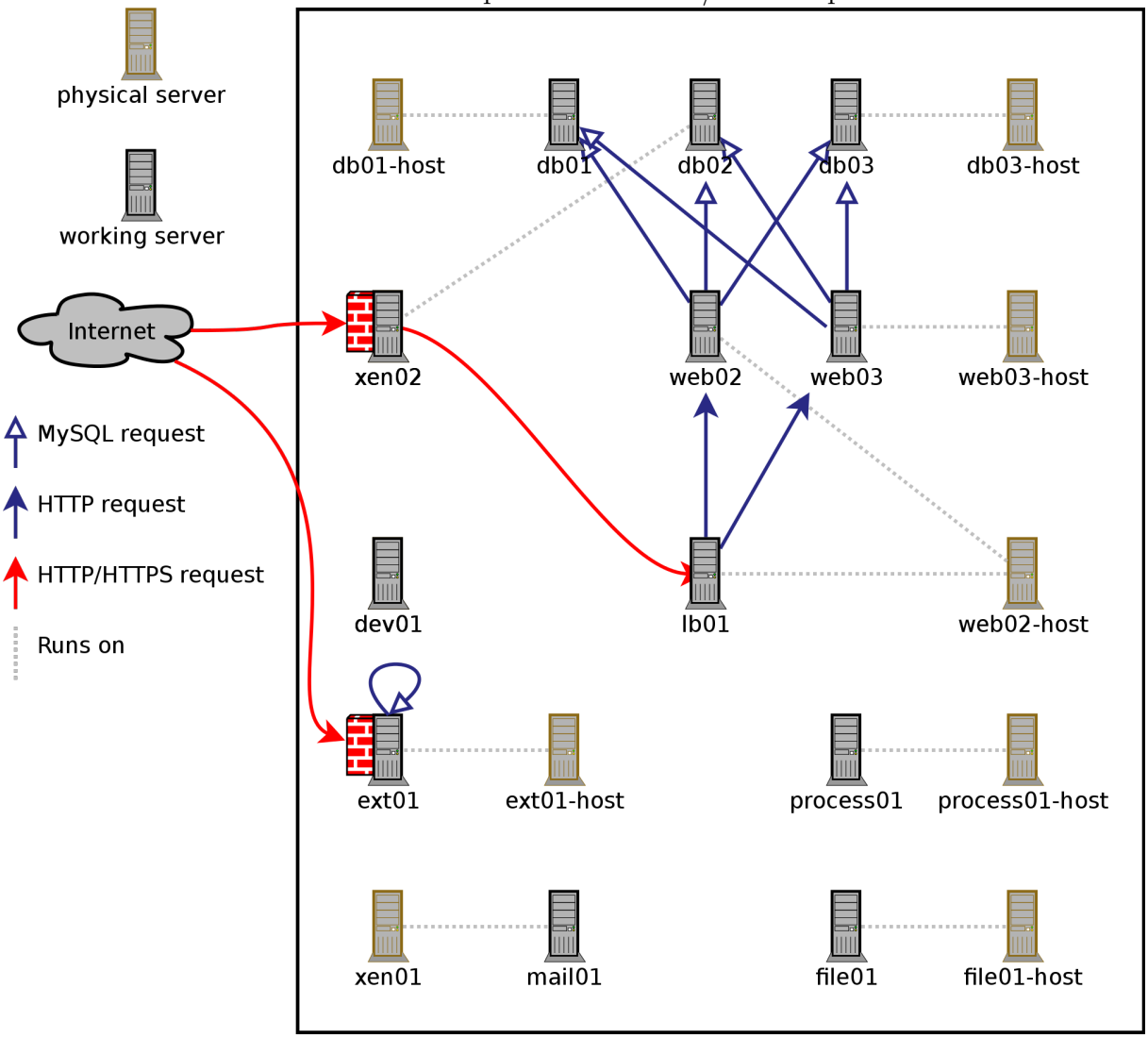
Příloha A

Obrázky

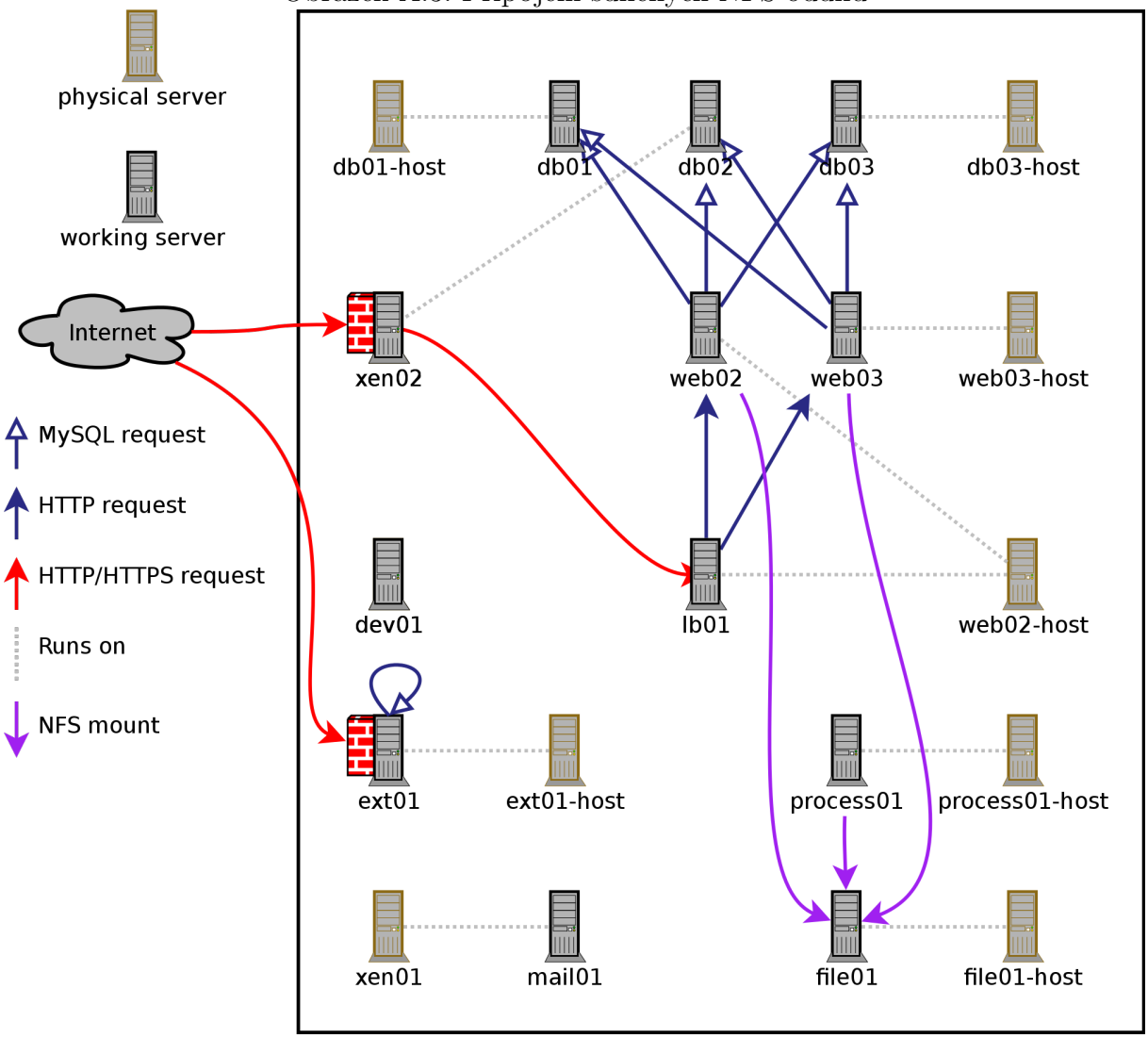
Obrázek A.1: Původní produkce s vyznačenými verzemi Operačních Systémů



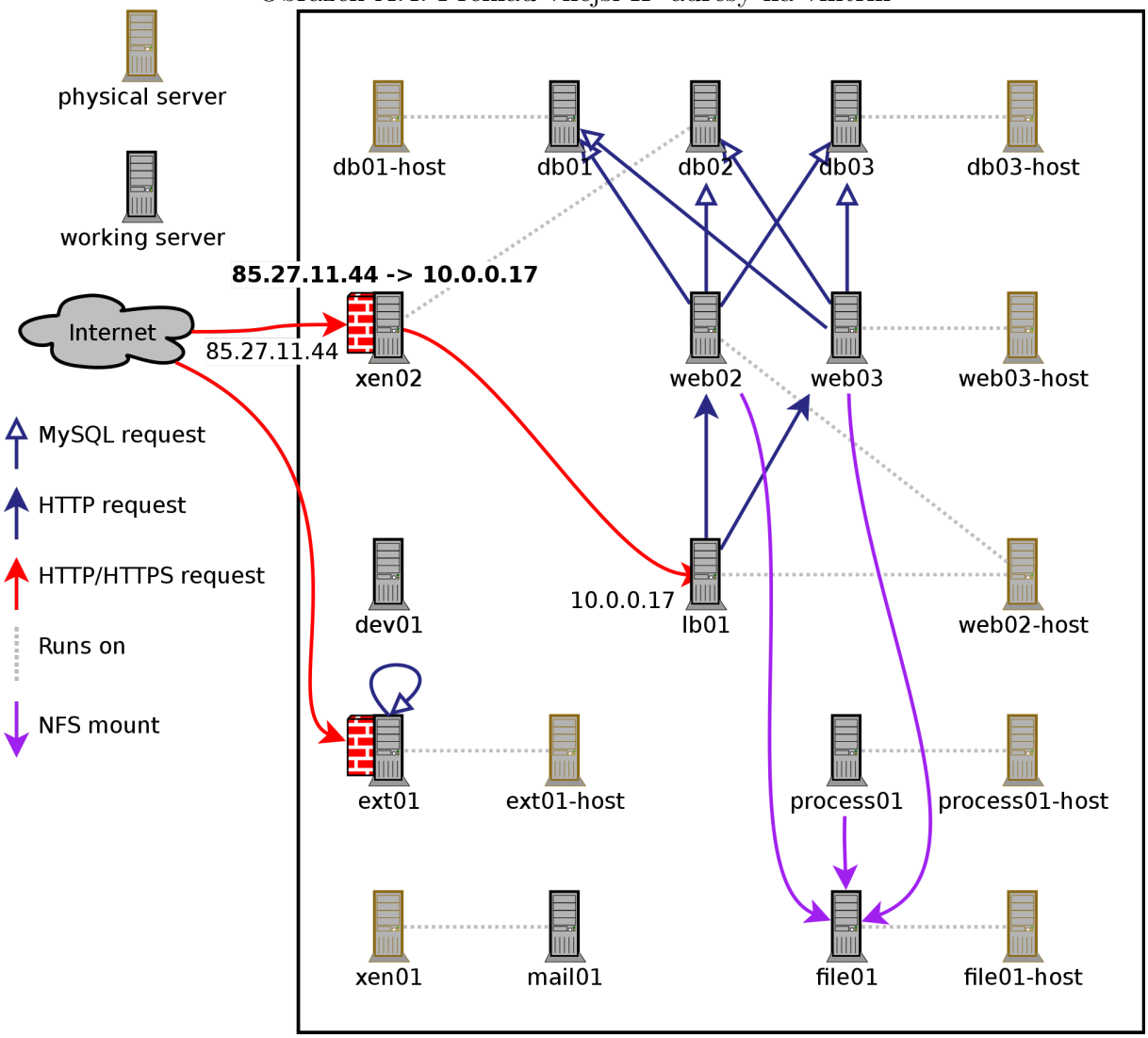
Obrázek A.2: Schéma zpracování HTTP/HTTPS požadavku



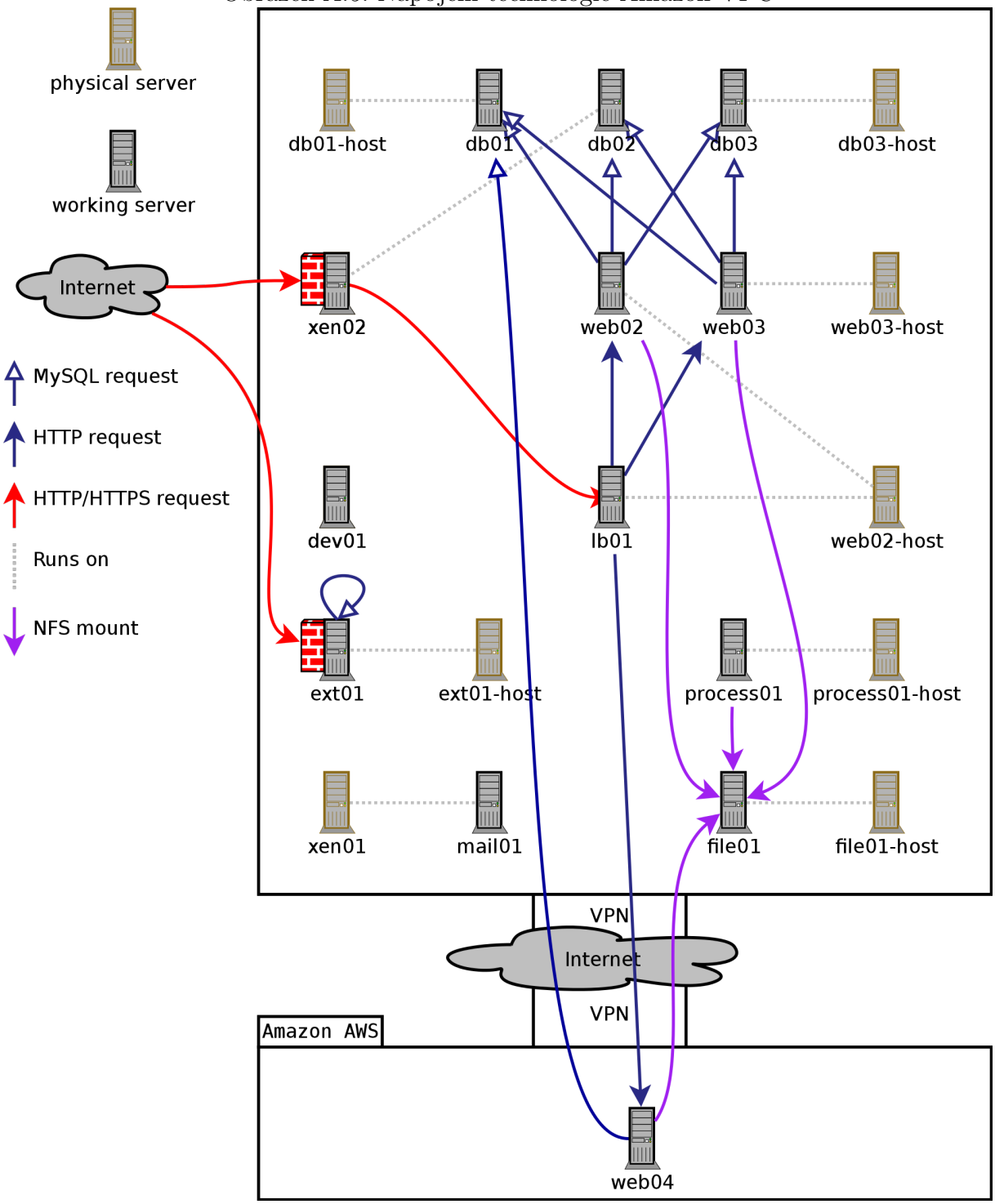
Obrázek A.3: Připojení sdílených NFS oddílů



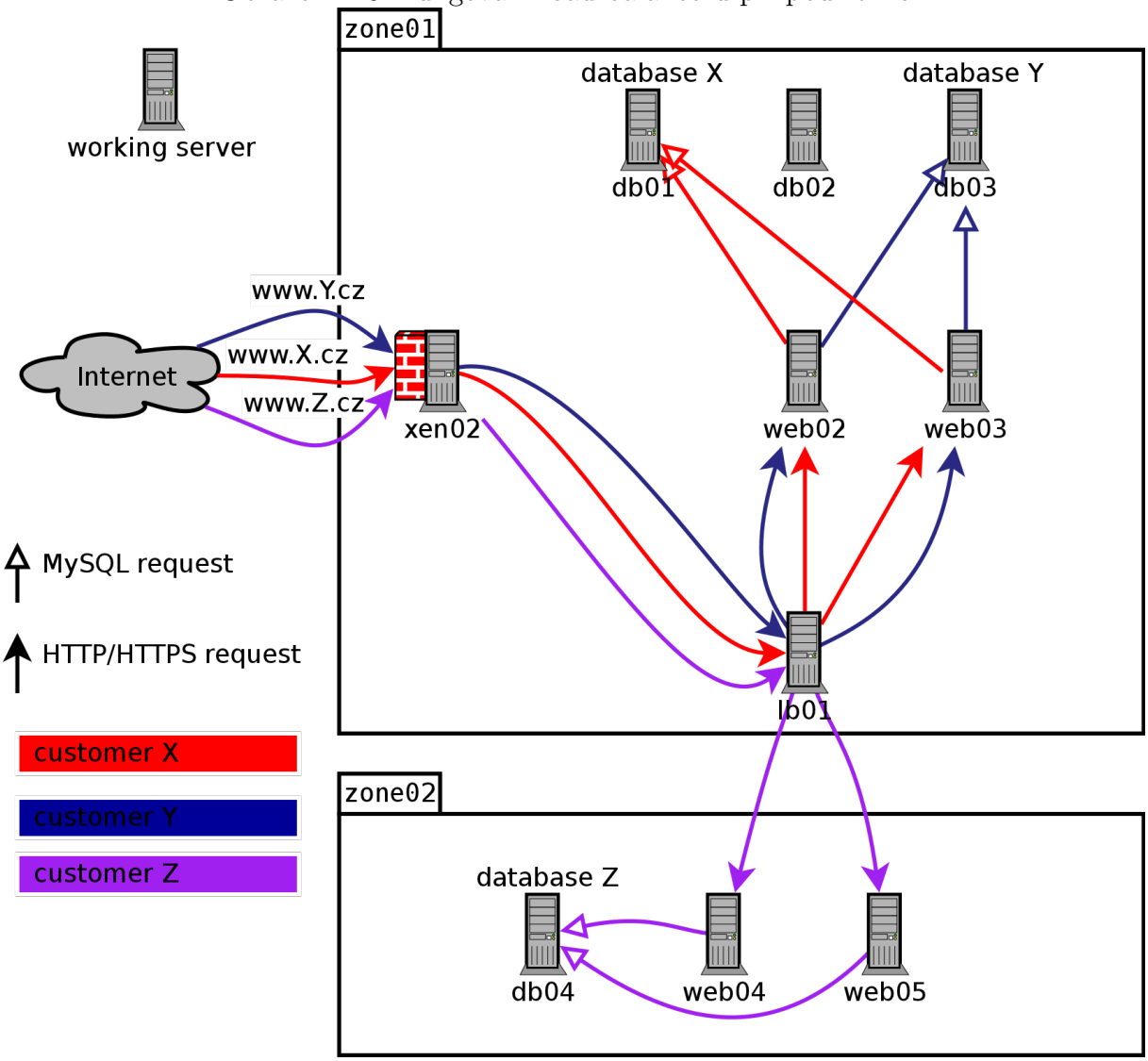
Obrázek A.4: Překlad vnější IP adresy na vnitřní



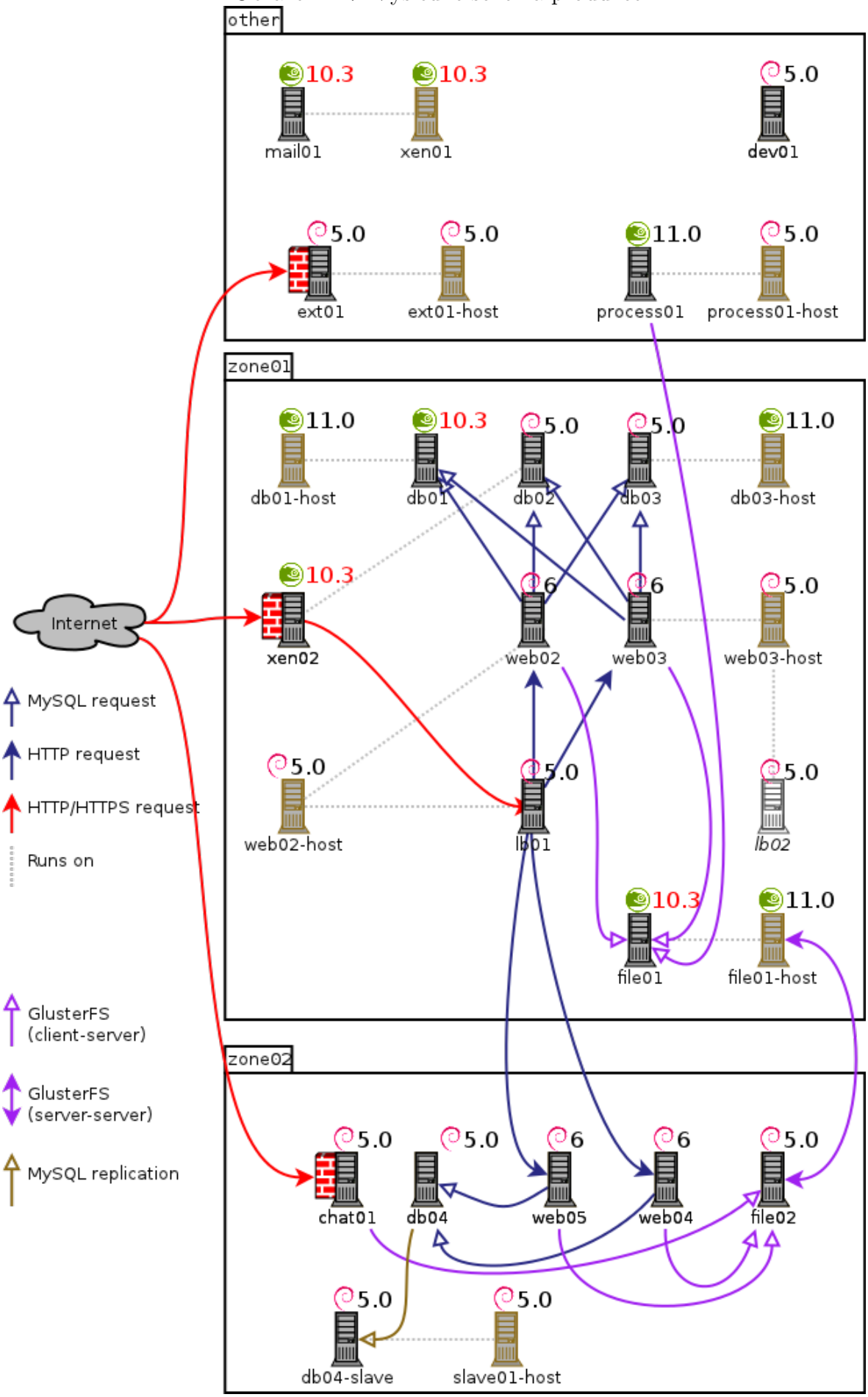
Obrázek A.5: Napojení technologie Amazon VPC



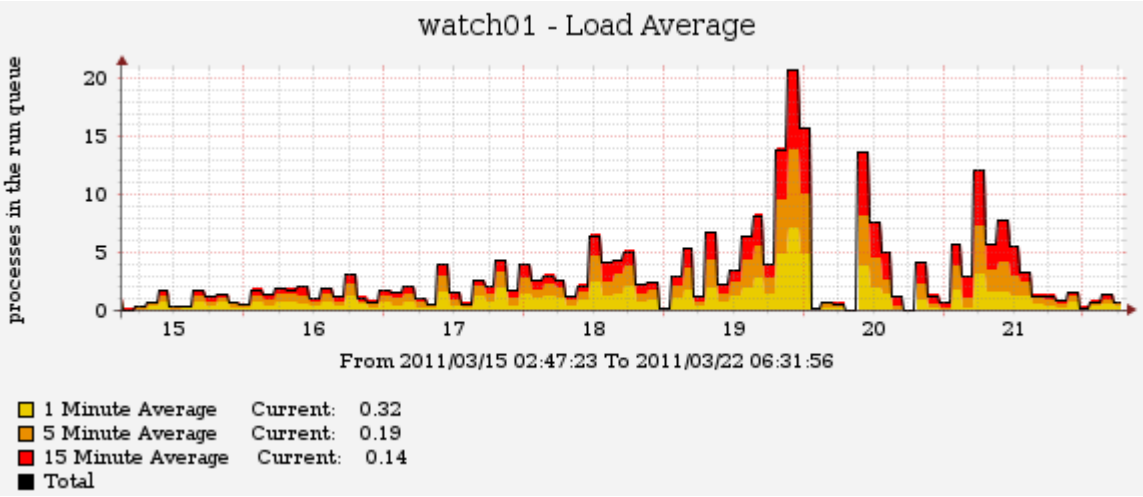
Obrázek A.6: Fungování load-balanceru při použití zón



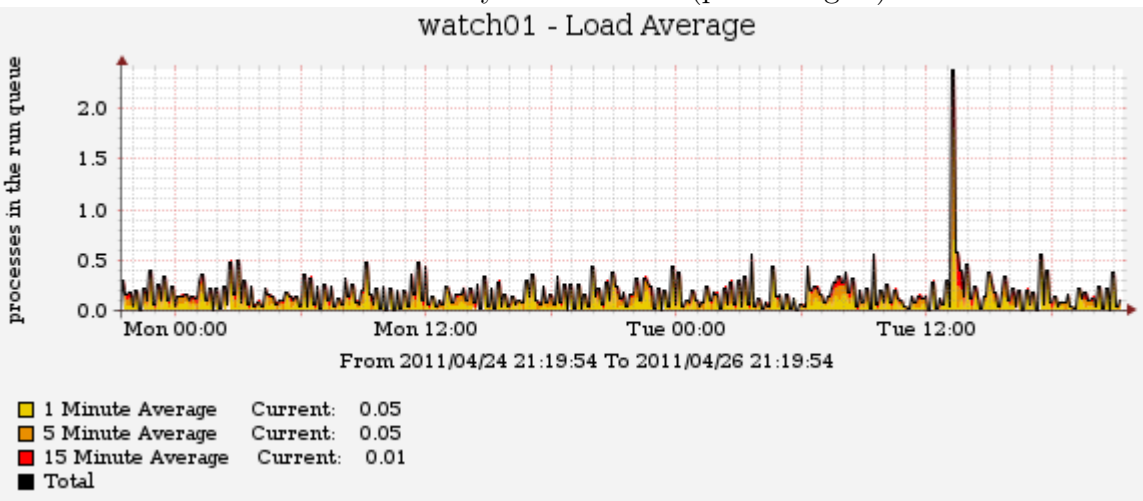
Obrázek A.7: Výsledné schéma produkce



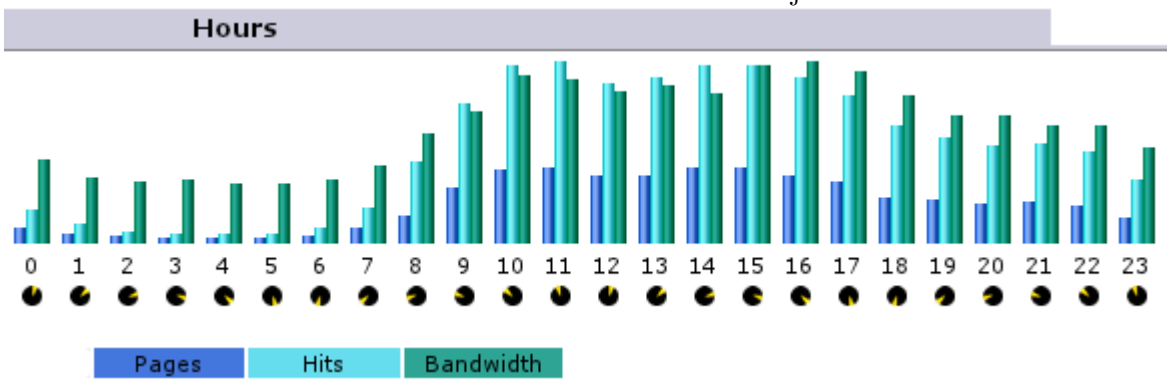
Obrázek A.8: Vytížení serveru (nagios + cacti)



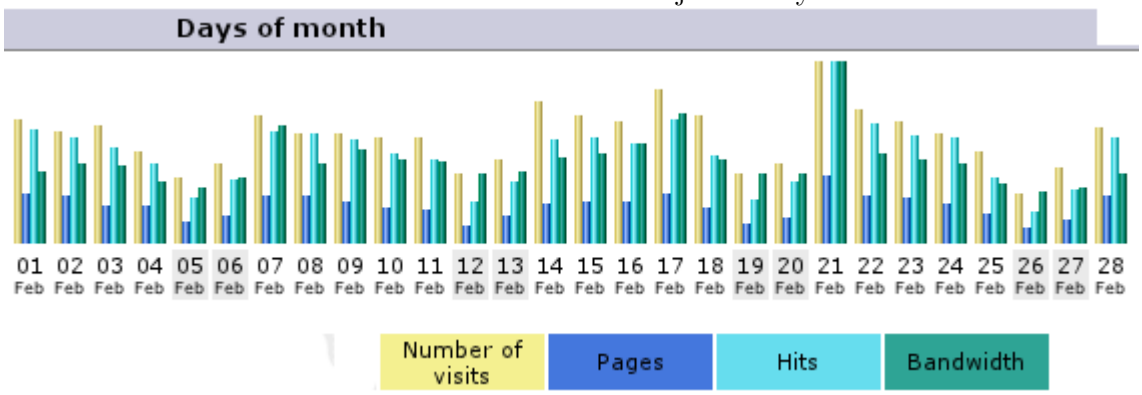
Obrázek A.9: Vytížení serveru (pouze nagios)



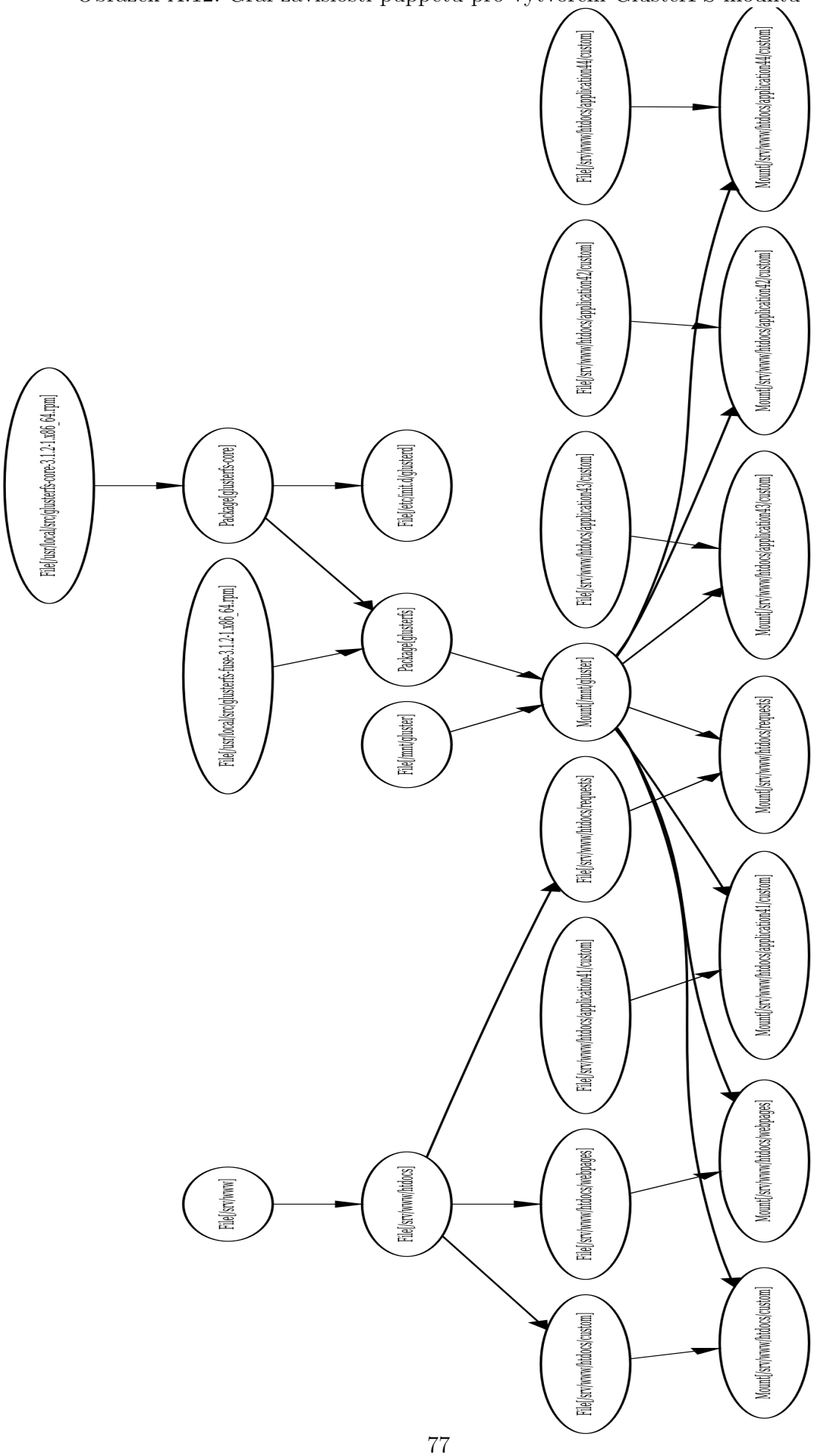
Obrázek A.10: Rozložení návštěvnosti v jednom dni



Obrázek A.11: Rozložení návštěvnosti v jednotlivých dnech měsíce



Obrázek A.12: Graf závislostí puppetu pro vytvoření GlusterFS mountů



Příloha B

Zdrojové kódy

B.1 IP SNMP scanner

```
1 #!/usr/bin/python
2
3 import os, sys, getpass, subprocess, re, socket, datetime
4
5 SERVERS = [ 'xen02', 'db01', 'db02', ] # Vycet serveru je jen ukazkovy
6 TMP_FILE = '/tmp/trac_ip.txt'
7 TRAC_ROOT = '/var/lib/trac/test'
8 COMMUNITY = 'XXX'
9 data=dict()
10
11 class SnmpScanner:
12     data = dict()
13     INTERFACE_NAME = 'interface_name'
14     IP = 'ips'
15     DNS = 'dns'
16
17     def find_interfaces(self):
18         for server in SERVERS:
19             self.data[server] = dict()
20             shell = subprocess.Popen("snmpwalk -c "+COMMUNITY+" -v 2c "+server+" ifName",
                   shell=True, stdout=subprocess.PIPE)
21             mib_table = shell.communicate()[0].split("\n")
22             for mib_record in mib_table:
23                 match_object = re.search('ifName.(\d*) = STRING: (\S*)$', mib_record, re.
                       MULTILINE)
24                 if match_object != None:
25                     interface_id = match_object.group(1)
26                     interface_name = match_object.group(2)
27                     self.data[server][interface_id] = dict()
28                     self.data[server][interface_id][self.INTERFACE_NAME] = interface_name
29
30     def find_dns(self):
31         for server in SERVERS:
32             for interface_id, interface in self.data[server].items():
33                 try:
34                     for ip in interface[self.IP]:
35                         try:
36                             hostname = socket.gethostbyaddr(ip)
37                             self.data[server][interface_id][self.IP][ip] = hostname[0]
```

```
38                          except:
39                              pass
40                              #self.data[server][interface_id][self.DNS] = "---"
41                  except:
42                      interface[self.IP] = dict()
43
44
45      def find_ips(self):
46          for server in SERVERS:
47              if server not in self.data: # TODO: ugly, make better
48                  self.data[server] = dict()
49              shell = subprocess.Popen("snmpwalk -c " + COMMUNITY + " -v 2c " + server + " ipAddressIfIndex.ipv4", shell=True, stdout=subprocess.PIPE)
50              mib_table = shell.communicate()[0].split("\n")
51              for mib_record in mib_table: #re.finditer('\S*$', tmp2, re.MULTILINE):
52                  match_object = re.search('ipv4."([^"]*)" = INTEGER: (\d*)$', mib_record, re.MULTILINE)
53                  if match_object != None:
54                      i = match_object.group(2)
55                      if str(i) not in self.data[server]: # TODO: ugly, make better
56                          self.data[server][str(i)] = dict()
57                      if self.IP not in self.data[server][str(i)]: # TODO: really ugly, make better
58                          self.data[server][str(i)][self.IP] = dict()
59
60                      self.data[server][str(i)][self.IP][match_object.group(1)] = ""
61
62      def print_info(self):
63          print("= Experimental IP list =")
64          print("This page is generated '''automatically'''. Do not modify it. Your changes will be overwritten.")
65          print("The ''localhost'' record refers to ''xen02''.")
66          print("List is generated by ative IP scanner. So it maps ''actual'' state of servers. Not the one in configurations.")
67          print("This page refers to state at " + str(datetime.datetime.now()))
68
69          wiki_page = ""
70          for server_name, server in self.data.items():
71              wiki_page += "== "+server_name+" ==\n"
72              for interface_id, interface in server.items():
73                  wiki_page += "||'''"+ (interface[self.INTERFACE_NAME] if self.INTERFACE_NAME in interface else "---") + "'''||"
74                  for ip, dns in interface[self.IP].items():
75                      #if self.data[server][self.INTERFACE_NAME] != "" and interface != "lo":
76                      wiki_page += "'''" + ip + "'''"
77                      if dns != "":
78                          wiki_page += " = " + dns
79                      wiki_page += ", "
80                          #wiki_page += str(self.data[server][interface]['host'][0])+" ||\n"
81                  wiki_page += " ||\n"
82          print wiki_page
83
84
85  obj = SnmpScanner()
86  obj.find_interfaces()
87  obj.find_ips()
88  obj.find_dns()
```

```
obj.print_info()
```

Listing B.1: IP SNMP scanner

B.2 site.pp

```
File { ignore => ['.svn', '.git'] } # Do not bother with those filename patterns when solving ensure=>directory, recurse=>true
Exec { path => "/usr/bin:/usr/sbin/:/bin:/sbin" } # This sets the path variable globally. No more path definitions in classes/nodes

exec { "apt-get update": # Update packages list
    alias => "aptgetupdate",
    refreshonly => true;
}

case $operatingsystem { # Special settings for different distributions
    OpenSuSE: {
        Package { provider => yum }

        case $operatingsystemrelease { # Special settings for different distribution releases
            "10.3": {
                yumrepo {
                    "puppet_repo":
                        descr   => "puppet repository for SLE_10",
                        enabled => 1,
                        baseurl => "http://download.opensuse.org/repositories/system:/management/SLE_10/"; # puppetd 0.25.5
                    "paderborn_10_3":
                        descr   => "paderborn-openSUSE-10.3",
                        enabled => 1,
                        baseurl => "ftp://ftp.uni-paderborn.de/pub/linux/opensuse/distribution/10.3/repo/oss/suse/";
                }
            }
            "11.0": {
                yumrepo {
                    "puppet_repo":
                        descr    => "puppet repository for openSUSE_11.1",
                        baseurl  => "http://download.opensuse.org/repositories/system:/management/SLE_11/",
                        enabled  => 1,
                        gpgcheck => 0;
                    "base":
                        descr    => "OpenSuSE 11.1",
                        baseurl  => "http://download.opensuse.org/distribution/11.1/repo/oss/suse/", # puppetd 0.25.4 (and many other packages)
                        enabled  => 1,
                        gpgcheck => 0;
                    "updates":
                        descr    => "OpenSuSE 11.1 updates",
                        baseurl  => "http://download.opensuse.org/update/11.1/",
                        enabled  => 1,
                        gpgcheck => 0;
                }
```

```
44                }
45            }
46        }
47 }
48
49 $puppetized_file="/tmp/puppetized_${operatingsystem}_${operatingsystemrelease}"
50 file { $puppetized_file: # This is just a way to mark servers as puppetized
51     ensure => present
52 }
53
54 import "nodes" # Get node definitions
```

Listing B.2: site.pp

Příloha C

Ostatní

C.1 Slovník pojmů jazyka puppet (anglicky)

As developers have found time and time again, terminology is critical to successful projects. The Puppet community has begun to coalesce around certain terms found to be useful in working with Puppet:

catalog A catalog is the totality of resources, files, properties, etc, for a given system.

class A native Puppet construct that defines a container of resources, such as File resources, Package resources, User resources, custom-defined resources (see also defined type), etc. A class can inherit from another class using the inherits keyword, and can also declare other classes.

agent or agent node An operating system instance managed by Puppet. This can be an operating system running on its own hardware or a virtual image.

declare (v.) To state that a class or a resource should be included in a given configuration. Classes are declared with the include keyword or with the class "foo": syntax; resources are declared with the lowercase file "/tmp/bar": syntax.

define (v.) To specify the contents and behavior of a class or defined resource type.

defined resource type or defined type (older usage: definition) A Puppet resource type defined at the application level. Defined types are created in the Puppet language and are analogous to macros in some other languages. Contrast with native type.

fact A detail or property returned by Facter. Facter has many built-in details that it reports about the machine it runs on, such as hostname. Additional facts can easily be returned by Facter (see Adding Facts). Facts are exposed to your Puppet manifests as global variables.

manifest A configuration file written in the Puppet language. These files should have the .pp extension.

module A collection of classes, resource types, files, and templates, organized around a particular purpose. See also Module Organisation.

native type A type written purely in Ruby and distributed with Puppet. Puppet can be extended with additional native types, which can be distributed to agent nodes via the pluginsync system. See the documentation for list of native types.

node (general noun) An individual server; for the purposes of discussing Puppet, this generally refers to an agent node.

node (Puppet language keyword) A collection of classes and/or resources to be applied to the agent node whose unique identifier ("certname") matches the specified node name. Nodes defined in manifests allow inheritance, although this should be used with care due to the behavior of dynamic variable scoping.

parameter (custom type and provider development) A value which does not call a method on a provider. Eventually exposed as an attribute in instances of this resource type. See Custom Types.

parameter (defined types and parameterized classes) One of the values which can be passed to this class or instances of this resource type upon declaration. Parameters are exposed as resource or class attributes.

parameter (external nodes) A global variable returned by the external node classifier.

pattern colloquial community expression for a collection of manifests designed to solve an issue or manage a particular configuration item, for example an Apache pattern.

plugin, plugin types a Puppet term for custom types created for Puppet at the Ruby level. These types are written entirely in Ruby and must correspond to the Puppet standards for custom-types.

plusignment operator An operator that allows you to add values to resource parameters using the +¿ ('plusignment') syntax. Available since version 0.23.1:

property (custom type and provider development) A value which calls a method on a provider. Eventually exposed as an attribute in instances of this resource type. See Custom Types.

provider A simple implementation of a type; examples of package providers are dpkg and rpm, and examples of user providers are useradd and netinfo. Most often, providers

are just Ruby wrappers around shell commands, and they are usually very short and thus easy to create.

realize a Puppet term meaning to declare a virtual resource should be part of a system's catalog. See also virtual resources.

resource an instantiation of a native type, plugin type, or definition such as a user, file, or package. Resources do not always directly map to simple details on the client — they might sometimes involve spreading information across multiple files, or even involve modifying devices.

resource object A Puppet object in memory meant to manage a resource on disk. Resource specifications get converted to these, and then they are used to perform any necessary work.

resource specification The details of how to manage a resource as specified in Puppet code. When speaking about resources, it is sometimes important to differentiate between the literal resource on disk and the specification for how to manage that resource; most often, these are just referred to as resources.

subclass a class that inherits from another class. Subclasses are useful for expanding one logical group of resources into another similar or related group of resources. Subclasses are also useful to override values of resources. For instance, a base class might specify a particular file as the source of a configuration file to be placed on the server, but a subclass might override that source file parameter to specify an alternate file to be used. A subclass is created by using the keyword inherits:

templates templates are ERB files used to generate configuration files for systems and are used in cases where the configuration file is not static but only requires minor changes based on variables that Puppet can provide (such as hostname). See also distributable file.

template class template classes define commonly used server types which individual nodes inherit. A well designed Puppet implementation would likely define a baseclass, which includes only the most basic of modules required on all servers at the organization. One might also have a genericwebserver template, which would include modules for apache and locally manageable apache configurations for web administrators.

Template classes can take parameters by setting them in the node or main scope. This has the advantage, that the values are already available, regardless of the number of times and places where a class is included:

This structure maps directly to a external node classifier and thus enables a easy transition.

type abstract description of a type of resource. Can be implemented as a native type, plug-in type, or defined type.

variable variables in Puppet are similar to variables in other programming languages. Once assigned, variables cannot be reassigned within the same scope. However, within a sub-scope a new assignment can be made for a variable name for that sub-scope and any further scopes created within it:

virtual resource a Puppet term for an resource that is defined but will not be made part of a system's catalog unless it is explicitly realized. See also realize.

```
class apache {
    service { "apache": require => Package["httpd"] }
}

class apache-ssl inherits apache {
    # host certificate is required for SSL to function
    Service[apache] { require +> File["apache.pem"] }
}

node mywebserver {
    include genericwebserver
}

node mywebserver {
    $web_fqdn = 'www.example.com'
    include singledomainwebserver
}

class ClassB inherits ClassA { ... }

$myvariable = "something"
```

Note that there are certain seemingly built-in variables, such as $hostname. These variables are actually created by Facter. Any fact presented by Facter is automatically available as a variable for use in Puppet.