Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Diplomová práce

BigCloud - backend pro veřejný cloudový systém

Bc. Petr Gregor

Vedoucí práce: Ing. Jiří Chludil

4. května 2015

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel dohodu, na základě níž se ČVUT vzdalo práva na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ust. § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisů.

V Praze dne 4. května 2015 .....

Odkaz na tuto práci

Gregor, Petr. *BigCloud - backend pro veřejný cloudový systém.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Cílem práce je navrhnout, implementovat a nasadit prototyp backendu cloudového systému. Tato písemná část se zabývá zejména návrhem architektury backendu a diskusí nad možnostmi jeho uskutečnění. Obsahuje také popis struktury výsledné implementace a řešení některých problémů, které při ní vznikly. Na konci textu naleznete návrh testovacího prostředí a dokumentaci API, určeného pro komunikaci s dalšími částmi systému.

Klíčová slova Cloud, IaaS, Xen, KVM, REST

Abstract

Goal of this thesis is to design, implement and deploy backend of prototype cloud system. This written portion mainly deals with design of backend architecture and discussion of different options for its implementation. It also contains description of final product's structure and solutions to some problems which arose during development. At the end of the text you can find design specification of testing environment and documentation of API, which is used communicate with other parts of the system.

Keywords Cloud, IaaS, Xen, KVM, REST

Obsah

Seznam obrázků

Úvod

Výpočetní technika. V dnešní době se s ní setkáme na každém kroku. Každý z nás vydává čím dál tím více času a peněz na její nákup, údržbu a provoz. Ať už se jedná o telefon, domácí počítač nebo firemní server, snažíme se vždy investovat pouze nutné minimum pro splnění našich požadavků. Jedním ze způsobů jak ušetřit, aniž bychom tyto požadavky kompromitovali, je výpočetní prostředky konsolidovat. Je levnější provozovat jedno petabytové datové úložiště než několik set malých lokálních úložišť. To samé platí pro jakoukoli výpočetní techniku. Konsolidovaný provoz je také efektivnější z pohledu nevyužité kapacity, která může být přerozdělena (například jinému zákazníkovi) a tak nakonec využita. Velkou výhodou je také flexibilita. Změna požadavků už nemusí znamenat novou investici či plýtvání, jenom přerozdělení zdrojů a případně rychlou změnu parametrů využívané služby.

Tyto výhody spolu s rozšířením dostupnosti a zvýšení přenosové kapacity komunikačních sítí vedou k popularizaci konceptu cloud computingu. Cloud computing je takové využití výpočetní techniky, při kterém jsou škálovatelné a pružné IT služby nabízeny zákazníkům za použití Internetu [1]. Typicky se tak prostředky provozovatele cloudu nacházejí v několika datacentrech, kde jejich hustota umožňuje efektivní správu a zabezpečení (záložní generátory, ostraha a tak dále).

Zadavatel této práce, firma S.I.C. spol. s.r.o. chce vyvinout své cloudové prostředí a následně ho nabídnout svým zákazníkům. Cílem této práce je navrhnout backend tohoto systému a následně implementovat a nasadit jeho prototyp do testovacího prostředí.

Kapitola 1

Požadavky

Detailní analýza požadavků koncových zákazníků není součástí této práce. V této kapitole se budu věnovat pouze popisu a přiřazení priorit požadavkům získaných přímo od zadavatele.

Požadavky rozděluji podle priority do kategorií „prototyp“ a „dodatečně“. Požadavky v kategorii „prototyp“ budou implementovány již v prototypu backendu. Ostatními požadavky se budu zabývat pouze okrajově a to proto, abych zajistil, že návrh systému umožní jejich pozdější zapracování.

1.1 Základní požadavky

R1: základní funkce backendu

Popis: Backend bude na základě příkazů z ostatních částí systému vytvářet a konfigurovat virtuální stroje (VM) a zajišťovat jejich chod.

Zdůvodnění: Backend bude sloužit jako nejnižší vrstva cloudového systému. Bude zajišťovat integraci s již existujícími virtualizačními technologiemi.

Priorita: prototyp

R2: minimální komplexita

Popis: Backend bude implementovat pouze pro zadavatele potřebné funkce. Vyhne se použití komplexních systémů (OpenStack, OpenNebula, ...)

Zdůvodnění: Čím je systém jednodušší, tím snazší je jeho údržba a rozvoj. Je také menší prostor pro vznik chyb a bezpečnostních mezer.

Priorita: prototyp

R3: application programming interface (API)

Popis: Backend bude implementovat HTTP REST API pro komunikaci s ostatními částmi systému.

Zdůvodnění: Jako nejnižší vrstva systému musí umožnit konzumaci svých služeb vrstvám vyšším. Hypertext transfer protocol (HTTP) representational state transfer (REST) API je zvoleno pro nízké náklady na implementaci a konzumaci a pro velký podíl na trhu existujících řešení.

Priorita: prototyp

R4: high availability (HA)

Popis: Backend bude vysoce dostupný. Výpadek jednoho stroje nesmí znemožnit provoz celého backendu.

Zdůvodnění: Selhání hardware je běžná reálie. Výpadek jednoho stroje nesmí vyřadit celý backend z provozu. Toto je důležité zejména pro management část backendu.

Priorita: dodatečně

R5: podpora hypervizorů

Popis: Backend umožní provoz virtuálních strojů na hypervizorech Xen a kernel virtual machine (KVM).

Zdůvodnění: Oba jsou významní hráči na poli hypervizorů. Každý je navržen jinak a může vynikat v jiné oblasti. Minimálně v prototypu je vhodné otestovat oba.

Priorita: prototyp

R6: skupiny hypervizorů

Popis: Backend bude pro vyšší vrstvy systému abstrahovat od jednotlivých výpočetních nodů. Dostupné prostředky bude nabízet ve formě skupin těchto výpočetních nodů. Skupiny budou

tvořeny nody vzájemně kompatibilními za účelem provádění vyvažování (viz R24).

Zdůvodnění: Vyšší vrstvy nemusí vědět, kde přesně virtuální stroj běží. Za zajištění jeho bezproblémového chodu zodpovídá backend.

Priorita: prototyp

1.2 Datová úložiště

R7: připojení disků

Popis: Backend umožní připojení virtuálních disků k VM. Virtuální disky jsou definované typem úložiště a svou velikostí.

Zdůvodnění: Různých druhů úložiště může být velké množství. Mohou se lišit svou rychlostí, maximální kapacitou a cenou. Pro různá použití jsou vhodné různé typy úložišť, a proto je nutné nechat výběr typu na klientovi.

Priorita: prototyp

R8: lokální datové úložiště

Popis: Backend umožní ukládat virtuální disky lokálně na výpočetních nodech.

Zdůvodnění: Jedná se o nejlevnější a nejjednodušší způsob uložení dat. Ideální pro dočasné prostory a testování.

Priorita: prototyp

R9: sdílené datové úložiště

Popis: Backend umožní ukládat virtuální disky na úložiště sdílené mezi alespoň dvěma výpočetními nody.

Zdůvodnění: Dostupnost stejných dat na více nodech zajistí kontinuitu provozu v případě selhání jednoho nodu.

Priorita: dodatečně

R10: clusterové úložiště

Popis: Backend umožní ukládat virtuální disky do clusterového úložiště (CEPH, GlusterFS, ...)

Zdůvodnění: Umožní oddělit výpočetní nody od úložiště a dále rozšíří možnosti live migrace.

Priorita: dodatečně

R11: úložiště obrazů cd a disků

Popis: Backend umožní práci s obrazy CD a disků uložených v externím NFS úložišti.

Zdůvodnění: Templaty disků a obrazy CD se běžně používají při instalaci a správě virtuálních strojů a velmi ji zjednodušují.

Priorita: prototyp

R12: změna velikosti disků

Popis: Backend umožní měnit velikost již vytvořených disků a to i za běhu virtuálního stroje.

Zdůvodnění: V průběhu používání virtuálního stroje je běžné narazit na nedostatek nebo přebytek prostoru.

Priorita: prototyp

R13: snapshoty

Popis: Backend umožní vytvářet snapshoty disků virtuálních strojů a jejich obnovu.

Zdůvodnění: Snapshoty umožňují rychlé uložení stavu virtuálního počítače a vrácení se k němu. Slouží k záloze před pokusem o provedení rizikové operace (například upgrade systému).

Priorita: dodatečně

1.3 Síť

R14: připojení do sítě

Popis: Backend umožní připojení virtuálních počítačů do sítí definovaných svým číslem VLAN a přidělenou IPv4 adresou.

Zdůvodnění: Síťová konektivita je základní službou. Sítí může být více a různých druhů (privátní, veřejné, rychlostně omezené, ...).

Priorita: prototyp

R15: zamezení použití nepřidělené ip adresy

Popis: Přístup virtuálního počítače do sítě s cizí nebo nepřidělenou adresou nebude umožněn.

Zdůvodnění: Zabrání nežádoucímu ovlivnění zbytku infrastruktury v případě pokusu o krádež identity nebo chybné konfiguraci stroje.

Priorita: prototyp

R16: IPv6

Popis: Podpora pro IPv6

Zdůvodnění: Umožnit přidělování a ochranu IPv6 adres stejným způsobem jako u IPv4.

Priorita: dodatečně

R17: DHCP

Popis: Backend bude přidělovat ip adresy pomocí DHCP.

Zdůvodnění: Usnadní konfiguraci virtuálních strojů.

Priorita: dodatečně

1.4 Správa virtuálních strojů

R18: vytvoření, smazání a změna parametrů

Popis: Backend umožní vytváření, mazání a úpravu parametrů virtuálních strojů. Konfigurovatelné parametry budou následující:

- Počet procesorů - ke kolika procesorům bude mít VM přístup
- RAM - kolik paměti RAM bude pro VM dostupné
- připojený obraz CD - jaký obraz disku použít pro simulaci CD mechaniky

- připojené disky (viz R7)
- připojené sítě (viz R14)

Skupina výpočetních nodů (R6) ve které VM poběží, bude specifikována při jeho vytvoření.

Priorita: prototyp

R19: změna skupiny hypervizorů

Popis: Backend umožní automatickou migraci VM mezi skupinami hypervizorů

Zdůvodnění: Výjimečně je třeba přesunout virtuální stroj do jiné skupiny z důvodu změny jeho parametrů nebo likvidace skupiny.

Priorita: dodatečně

R20: VNC konzole

Popis: Backend umožní připojení ke grafické konzoli na virtuálním stroji přes protokol VNC.

Zdůvodnění: Konzole se typicky používá v případě problémů s běžným administračním prostředkem (typicky ssh nebo rdp). Nezávisí na OS uvnitř virtuálního stroje a tudíž se od ní nelze konfigurační chybou „odříznout“.

Priorita: prototyp

R21: změna parametrů VM za běhu

Popis: Backend umožní změnu parametrů virtuálního stroje bez nutnosti tento restartovat. Pouze tam, kde to umožňuje použitý hypervisor.

Zdůvodnění: Umožní přidat rychlejší disk nebo další procesor při neočekávaném náporu na virtuální stroj, aniž by došlo k přerušení dodávané služby.

Priorita: dodatečně

R22: offline migrace

Popis: Backend umožní offline (VM musí být vypnutý) migraci na jiný výpočetní node.

Zdůvodnění: Offline migrace je proveditelná i bez sdíleného úložiště. Je důležitá pro vyvažování obsazeného výkonu a provádění údržby na výpočetních nodech.

Priorita: dodatečně

R23: live migrace

Popis: Backend umožní online (VM může být zapnutý) migraci na jiný výpočetní node.

Zdůvodnění: Live migrace vyžaduje sdílené úložiště. Je ale mnohem praktičtější, protože nevyžaduje žádné přerušení služby.

Priorita: dodatečně

R24: vyvažování zátěže

Popis: Backend bude automaticky měnit umístění VM na výpočetních nodech tak, aby maximalizoval využití hardware. Optimálně závisí na R23, ale omezeně lze použít i R22.

Zdůvodnění: Parametry a vytížení virtuálních strojů se mohou měnit a s tím může přijít přetížení nebo nevyužití jednotlivých výpočetních nodů. Backend musí zajistit průběžné vyvažování zátěže.

Priorita: dodatečně

1.5 API

R25: vytvoření, smazání a změna parametrů VM

Popis: API umožní vytvářet, mazat a měnit parametry virtuálních strojů. Parametry popsány v R18.

Priorita: prototyp

R26: aktuální stav virtuálního stroje

Popis: API pro každý VM poskytne informace o jeho aktuálním stavu (zapnuto, vypnuto) a umožní ho změnit (zapnout, vypnout, restartovat).

Zdůvodnění: Stav VM se může měnit bez zásahu z API (softwarové vypnutí stroje).

Priorita: prototyp

R27: dostupnost

Popis: API bude po síti dostupné zbytku systému.

Zdůvodnění: Ostatní části systému nemusí být na stejném serveru.

Priorita: prototyp

R28: informace o dostupných skupinách výpočetních nodů

Popis: API bude ostatní části systému informovat o dostupných skupinách výpočetních nodů (R6).

Zdůvodnění: Ostatní části systému budou používat API pro získání seznamu aktuálně dostupných výpočetních prostředků.

Priorita: prototyp

R29: informace o aktuálním vytížení skupin výpočetních nodů

Popis: API bude ostatní části systému informovat o aktuálním zatížení jednotlivých skupin a dostupných volných prostředcích (RAM, CPU load)

Zdůvodnění: Ostatní části systému budou používat API pro získání aktuálního stavu výpočetních prostředků.

Priorita: dodatečně

R30: informace o dostupných úložištích pro každou skupinu nodů

Popis: API bude ostatní části systému informovat o úložištích, která jsou k dispozici každé skupině výpočetních nodů.

Zdůvodnění: Konzument API se musí dozvědět, jaká jsou platná úložiště pro vytváření nových disků. Souvisí s R7.

Priorita: prototyp

R31: informace o volném místě na úložištích

Popis: API bude ostatní části systému informovat, kolik volného prostoru je k dispozici na jednotlivých úložištích. Tedy jak maximálně velký virtuální disk na nich lze vytvořit.

Zdůvodnění: Některá úložiště mohou být přechodně přetížena a nemusí mít kapacitu pro vytvoření nového disku. Konzument API se to musí dozvědět.

Priorita: dodatečně

R32: informace potřebné k připojení k VNC konzoli

Popis: API pro každý VM poskytne informace potřebné k připojení k VNC konzoli (IP adresa, port, heslo)

Zdůvodnění: Díky přesouvání VM v systému (R24) se VNC endpoint může měnit. API musí poskytnout aktuální informace zbytku systému.

Priorita: prototyp

Kapitola 2

Návrh

V této kapitole popíšu některá již existující řešení cloudových systémů a navrhnu řešení své.

2.1 Existující řešení

Nabídka již existujících řešení je poměrně rozsáhlá. Několik jich zde krátce představím.

2.1.1 XenServer

Jedná se o nedávno open-sourcovaný, dříve uzavřený projekt společnosti Citrix. Staví na hypervizoru Xen, nad kterým pracuje toolstack xapi. Xapi zajišťuje komunikaci více výpočetních nodů, implementuje snapshoty, migraci, disaster recovery, integraci s Open vSwitch a další funkce. Navíc poskytuje XenAPI, které lze dále použít pro integraci s dalšími službami a administraci. Xapi je nainstalováno na každém výpočetním nodu a jeho databáze se synchronizuje mezi všemi nody v jedné skupině. Jeden z nodů je takzvaný master a jedině z něj může probíhat konfigurace skupiny. Na rozdíl od dalších v této sekci představených produktů se nejedná o příliš komplexní systém. Neobsahuje webový management, management storage, sítí, autentizační server a podobné další funkce, které jsou pro cloudové systémy samozřejmostí. [2] [3]

2.1.2 OpenStack

OpenStack je komplexní otevřený systém distribuovaný pod licencí Apache 2.0. Skládá se z mnoha částí, kde každá zajišťuje specifickou část infrastruktury (computing, storage, networking, dashboard, ...). Jednotlivé části mezi sebou komunikují pomocí REST API. OpenStack, respektive Compute, část OpenStacku zodpovědná za správu výpočetních prostředků, umožňuje integraci s mnoha hypervizory včetně Xenu, KVM, Hyper-V a dokonce i s kon-

teinerovými systémy jako je LXC nebo Docker. OpenStack je opravdu sofistikovaný systém, který se snaží mít řešení pro každou situaci. Celkem obsahuje více než 20 milionů řádků kódu a je vyvíjen společným úsilím stovek společností. Mezi nimi jsou i velcí hráči informačního průmyslu jako je PayPal, RackSpace nebo Yahoo!. [4] [5]

2.1.3 OpenNebula

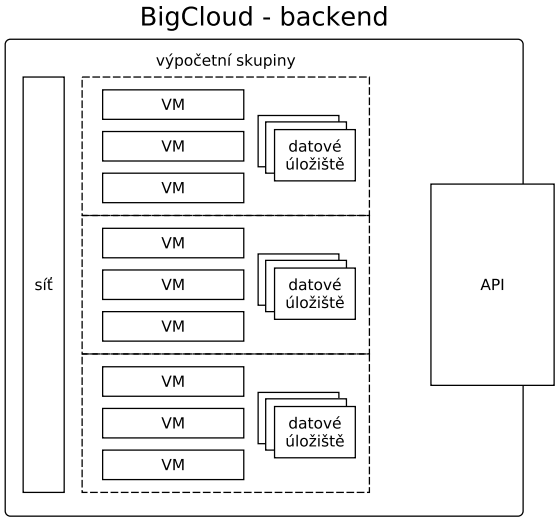
Oproti OpenStacku se OpenNebula zaměřuje spíše na interní (firemní) cloudy [6]. Je také menší a snáze nasaditelná (jádro není rozdělené do mnoha různých projektů). Navzdory tomu obsahuje velké množství funkcí a řeší mnoho různých situací. OpenNebula umožňuje provozovat virtuální stroje na hypervisorech KVM, Xen a VMWare ESX a vCenter. Stejně jako OpenStack je distribuována pod licencí Apache 2.0. [7]

2.2 Návrh architektury

Při návrhu jsem postupoval od vnějšího pohledu na backend přes dekompozici k detailnější analýze jeho jednotlivých součástí (top-down návrh).

2.2.1 Povrchní pohled

Na diagramu 2.1 vidíte backend, jak vyplývá z požadavků zadavatele. Pro využívání jeho služeb nejsou další informace nutné. Pro efektivní návrh je ale třeba provést dekompozici.

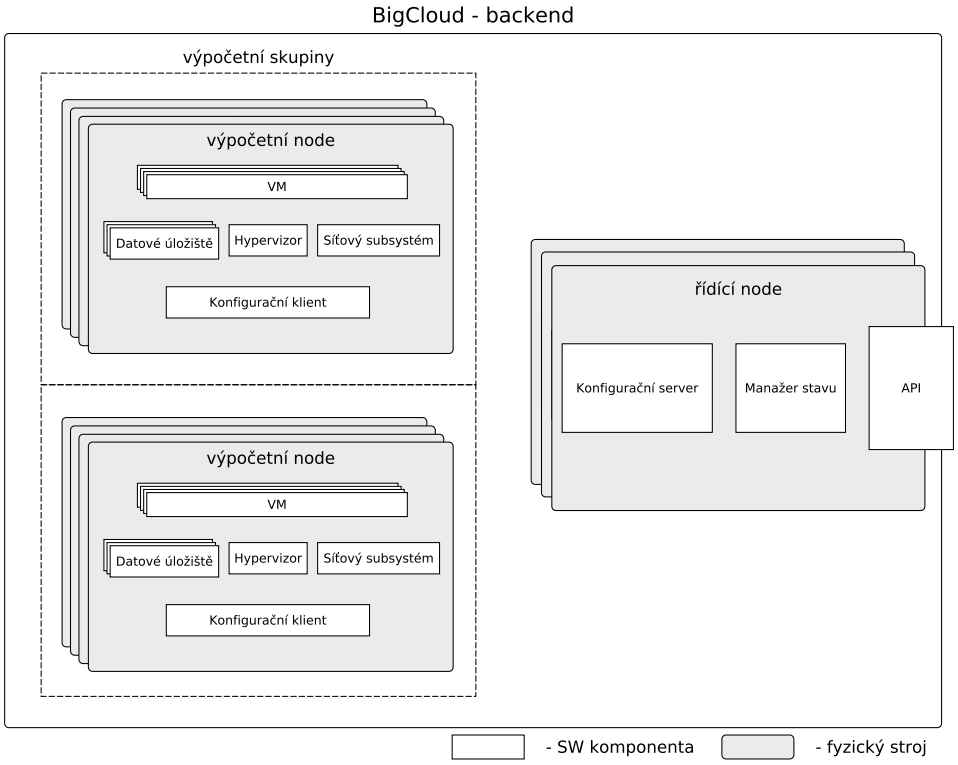


Obrázek 2.1: Vnější pohled na backend

2.2.2 Detailní pohled

Diagram 2.2 ukazuje backend po dekompozici. Zanesl jsem do něj všechny komponenty, které systém vyžaduje k chodu. Explicitně jsem také oddělil výpočetní nody, tedy stroje, na kterých poběží VM klientů od řídících nodů, což jsou stroje, kde poběží pouze software zajišťující chod backendu. Tento návrh zaručí, že jakékoli problémy způsobené selháním hardware, nedostatkem výkonu nebo útokem na virtuální stroje neovlivní administrační schopnosti backendu jako celku.

Z důvodu zjednodušení údržby a implementace požadavku R4 jsem se rozhodl navrhnout řídící node tak, aby bylo bez problémů možné provozovat několik jeho kopií zároveň. Data mezi nimi budou synchronizována a libovolný z nich bude možné použít pro přístup k API.



Obrázek 2.2: Detailní pohled na backend

Následuje popis jednotlivých komponent a jejich zodpovědností.

API

Vstupní bod do systému. Umožňuje po síti měnit a získávat aktuální stav systému, jak je uložen u manažera stavu.

Manažer stavu

Udržuje aktuální stav backendu a slouží jako jeho „zdroj pravdy“. Spravuje také veškerá data sesbíraná z výpočetních nodů a zajišťuje běh periodických procesů, jako je automatické vyvažování zátěže. Manažera stavu využívá API a konfigurační server.

Konfigurační server

Zajišťuje distribuci konfiguračních dat od manažera stavu k jednotlivým výpočetním nodům, respektive k jejich konfiguračním klientům. Jeho další funkce je přenos informací (vytížení, aktivní VM, ...) získaných na výpočetních nodech opačným směrem.

Konfigurační klient

Přijímá příkazy z konfiguračního serveru a na jejich základě konfiguruje výpočetní node. Informace o výsledku operací a statistiky výkonu odesílá zpět konfiguračnímu serveru.

Datové úložiště

Blokové úložiště pro použití virtuálními stroji. Slouží jako backend virtuálních disků.

Síťový subsystém

Zajišťuje přístup k síti pro virtuální stroje. Slouží jako backend virtuálních síťových adaptérů.

Hypervizor

Software zajišťující běh virtuálních strojů a emulaci jejich periferií.

Virtuální stroj

Izolované prostředí sloužící pro běh počítačových systémů. Jedná se o finální produkt celého projektu.

2.3 Výběr software

Začínat z ničeho není v dnešní době téměř nikdy správným řešením. V této sekci vyberu již existující otevřená řešení, na kterých postavím jednotlivé komponenty systému.

2.3.1 distribuce GNU/Linuxu

Použití operačního systému GNU/Linux vyplývá už z požadavku R5. KVM totiž není s jiným OS kompatibilní, je součástí Linuxového jádra. Distribucí Linuxu je ale celá řada.

Pro všechny stroje v tomto systému budu používat Debian. Tato distribuce je oblíbená, stabilní, má obsáhlé softwarové repozitáře a mám s ní zkušenosti. Návrh ale není na použití Debianu nijak závislý, změnit distribuci by v případě potřeby nebyl velký problém.

2.3.2 API a manažer stavu

Tyto dvě úzce provázané a velmi specifické části systému jsem se rozhodl sloučit a jako jediné skoro celé implementovat. Přesto nebudu začínat úplně od nuly. Budu stavět na základech frameworku určeného pro vývoj REST API a vhodného pro implementaci asynchronního a vysoce dostupného manažera stavu.

Kandidátů je celá řada. V poslední době velké množství projektů nasazuje REST na místo dříve populárních SOAP služeb. Nástroje s tímto trendem drží krok a tak je framework pro budování REST služeb dostupný pro velké množství jazyků a platforem [8] [9] [10].

Já jsem se rozhodl pro framework Hapi na platformě Node.JS. Jedná se o stabilní, spolehlivý a otevřený software používaný velkými firmami po celém světě[11]. Vývoj začal a je stále veden výzkumnou divizí amerického Wallmartu. Hapi mi ušetří práci mimo jiné s routingem, serializací, validací, logováním a testováním. V mém rozhodnutí hrají významnou roli také mé zkušenosti s tímto systémem. Nebudu se ho muset učit úplně od nuly.

Platforma Node.JS je postavená nad JavaScriptovým enginem V8[12]. To znamená, že všechen kód budu psát v jazyce JavaScript a budu používat asynchronní I/O model (více v sekci 3.3.1). Také to znamená, že budu stavět na prověřeném enginu používaném miliony lidí po celém světě, který interpretuje jasně standardizovaný jazyk [13].

Databáze

Součástí manažera stavu musí být také persistentní úložiště, kam se bude stav zapisovat. Za tímto účelem použiji již existující databázový systém. Mé požadavky jsou zejména otevřenost, stabilita, dobrá podpora pro vysokou dostupnost a kompatibilita s Node.JS. Ze zavedených[14] databází splňují mé

požadavky PostgreSQL, MongoDB a MySQL. Všechny jsou připravené pro produkční prostředí, otevřené a kompatibilní s Node.JS[15] [16] [17]. Velmi se ale liší v podpoře vysoké dostupnosti.

MySQL umožňuje zabránit výpadkům několika způsoby [18]. Pro mě jsou zajímavé dva z nich a to MySQL Cluster a distributed replicated block device (DRBD). MySQL Cluster je upravená verze MySQL se speciálním tabulkovým enginem NDB. Umožňuje s několika omezeními synchronní replikaci dat, bohužel pouze v rámci lokální sítě [19]. Druhou možností je provozovat MySQL nad DRBD úložištěm v active/standby módu. MySQL je aktivní jen na jednom stroji a DRBD zajišťuje replikaci dat na stand-by zálohy. V případě výpadku hlavního stroje se služba spustí na záložním [20]. Jedná se o netriviální řešení, při kterém kooperuje MySQL, DRBD, Pacemaker a Corosync. Nároky na konfiguraci a údržbu jsou tedy poměrně vysoké.

PostgreSQL řeší tento problém také vícero způsoby. Jako příklad mohu uvést integraci s DRBD nebo přeposílání transakčních logů na slave servery. Bohužel tyto i všechny ostatní vyžadují ke své optimální funkci externí nástroje a komplikovanou konfiguraci. [21]

Třetí možností je MongoDB. Na rozdíl od obou předchozích se nejedná o SQL databázi, ale o dokumentovou databázi. Při přístupu k ní se místo SQL používá speciální dotazovací jazyk založený na BSON dokumentech. Spolu s SQL chybí v MongoDB další funkce, na které jsme z tradičních databází zvyklí. Zejména jsou to transakce a spojování (joining) tabulek. Na druhou stranu nabízí integrované HA řešení použitelné i po komunikačních linkách s vysokou latencí, atomické zápisy v rámci jednoho dokumentu a vysokou míru flexibility, kterou přináší absence předem definovaných schémat.[22] [23]

Zejména díky výborné integrované podpoře pro HA jsem vybral MongoDB. Pro využití v rámci manažera stavu není absence relací a transakcí problém. Nahradím je vnořováním s využitím atomicity zápisu v rámci jednoho dokumentu.

2.3.3 Konfigurační server a klient

Automatickou vzdálenou konfiguraci lze řešit mnoha způsoby. Já se soustředím na software specificky navržený pro tuto činnost. Požaduji od něj otevřenost, stabilitu, rozšiřitelnost a možnost obousměrné komunikace.

Prvním kandidátem je **Ansible**. Tento nástroj pro konfigurační management je napsán v jazyce Python a namísto vlastního agenta na konfigurovaných strojích používá pouze ssh. Výhodou tohoto přístupu je zjednodušení správy celého systému a flexibilita. Nevýhodou je nemožnost informovat server o čemkoli asynchronně, vše si musí po připojení master vyžádat. Konfigurace Ansiblu se provádí pomocí takzvaných „playbooků“. Jsou to YAML soubory obsahující jednotlivé stavy nebo kroky procesu, které má Ansible vynutit/vykonat. Pro mé účely není vhodný zejména díky absenci asynchronní zpětné vazby. [24] [25]

Podobně jako Ansible využívá **Salt** ke svému běhu Python. Oproti němu se ale jedná o složitější systém, který kromě konfiguračního managementu zvládá například i získávat data ze vzdálených systémů nebo reagovat na události. Salt využívá klient-server model, kde se všechny konfigurované stroje (minions) připojují k jednomu nebo několika konfiguračním serverům (masterům). Komunikace v Saltu je založená na ZeroMQ a spojení zůstává dlouhodobě otevřené. Doba odezvy systému je tak mnohem kratší než při použití ssh (i to je ale v Saltu možné). Popis konfigurace v Saltu se skládá z sls souborů. V defaultním nastavení se jedná o YAML soubory zpracované Jinja2 preprocesorem, který umožňuje provádět templatování a automatické generování stavů. Salt také obsahuje event bus, který lze využívat z externích programů a posílat tak obousměrně zprávy v rámci Saltem spravované sítě. [26] [27] [28]

Dalšími známými nástroji jsou **Chef** a **Puppet**. Oba jsou napsané v Ruby a používají client-server architekturu. Ani jeden z nich ale nemá alternativu ke komunikační sběrnici implementované v Saltu, kterou bych pro implementaci potřeboval. [29] [30]

Z průzkumu mi jako jasná volba vyšel Salt a to zejména díky své otevřené komunikační sběrnici umožňující rychlou komunikaci mezi servery. Jeho použitelnost pro backend cloudového systému dokazuje i existence Salt Virtu, což je na KVM zaměřený cloudový controller postavený na Saltu. [31]

I přes to, že mi Salt ušetří hodně práce, tak očekávám, že ho budu muset rozšířit o některé specifické funkce. Bude to nejméně kód pro sběr dat o stavu virtuálních strojů a komunikace s manažerem stavu.

2.3.4 Datové úložiště

Datových úložišť implementuji více, klient si vybere, který z dostupných typů se pro jeho použití hodí nejvíce. Od úložiště vyžaduji pouze jeho dostupnost jako blokové zařízení.

Velmi často používané a v Linuxových distribucích integrované je úložiště **LVM2**. Umožňuje sloučit několik blokových zařízení (physical volume) do jednoho celku (volume group) a ten flexibilně dělit na části. Tyto části (logical volume) jsou pak dostupné jako další blokové zařízení. Podporuje i COW snapshoty a změnu velikosti vytvořených oddílů. Typicky se do LVM2 přidávají bloková zařízení vytvořená pomocí softwarového RAID nebo jinak zabezpečené proti selhání. Samotné LVM2 nijak data nechrání.

Pokročilejší formu správy dat nabízí systém **ZFS**. Kromě filesystému v sobě integruje správu oddílů, podporu pro tvorbu blokových zařízení (ZVOL), COW snapshoty s možností je diferencovat a posílat po síti, RAID a hlavně ochranu dat před poškozením pomocí kontrolních součtů. Všechny tyto funkce mají ovšem svou cenu a tak je ZFS oproti LVM náročnější na systémové prostředky, zejména RAM.

V rámci prototypu implementuji tyto dva typy úložiště. V pozdějších fázích projektu ale určitě přibudou další.

2.3.5 Síťový subsystém

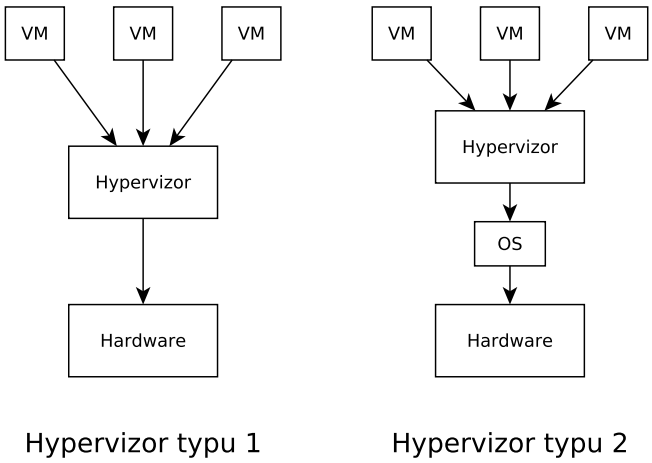
Pro řízení sítě v rámci hypervizoru jsem zvolil OpenVSwitch. Jedná se o alternativu ke klasickému Linux bridgi, který je implementován přímo v jádře. Oproti němu jde o systém navržený pro správu sítí virtuálních strojů. Nabízí funkce jako migraci stavu z jednoho serveru na jiný, RSPAN, netflow a další. Jedná se také o OpenFlow kompatibilní switch a tak bude v pozdější fázi projektu možné veškerou konfiguraci všech instancí řešit z jednoho místa (controlleru). [32] [33]

2.3.6 Hypervizor

Systém bude podle požadavku R5 implementovat podporu pro hypervizory Xen a KVM. Ne najednou, ale v rámci různých skupin hypervizorů.

Xen je jediný otevřený hypervisor typu 1. To znamená, že běží přímo na hardware (viz diagram 2.3). Není součástí operačního systému. Mezi jeho vlastnosti patří malávelikost (kolem 1MB), podpora pro spouštění VM v hardwarově akcelerovaném (HVM) i paravirtualizovaném (PV) režimu a schopnost oddělit ovladače zařízení od kontrolního VM (dom0). Podpora PV režimu umožňuje provozovat Xen i na strojích, které nedisponují hardwarem potřebným pro plnou virtualizaci (procesor s podporou VT-x nebo AMD-V). Xen je možné spravovat pomocí několika toolstacků. V současných verzích Xenu je za defaultní považovaný toolstack postavený nad knihovnou libxl. Ten také budu za pomoci jednoduchého příkazu xl používat k interakci s hypervizorem. [34]

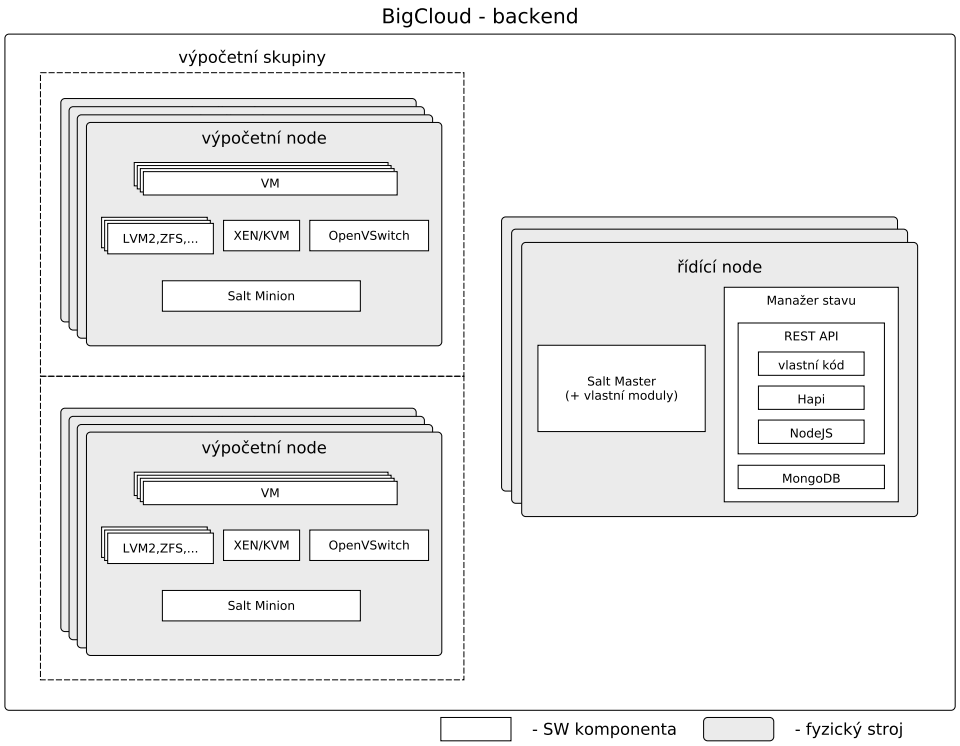
KVM je modul Linuxového kernelu, který umožňuje programům přístup k virtualizačním rozšířením moderních procesorů. Sám o sobě nemůže jako hypervizor fungovat, vyžaduje ještě alespoň qemu, což je emulátor vytvářející samotné prostředí, ve kterém virtuální počítač běží (Xen ho v HVM režimu používá také). Většinou se kromě KVM a qemu používá i libvirt, který usnadňuje správu virtuálních strojů. V mém prostředí ho ale vynechám a budu pracovat přímo s qemu (to poté interně používá KVM pro akceleraci virtualizace). [35]



Obrázek 2.3: Typy hypervizorů

2.4 Výsledný návrh

Na diagramu 2.4 si můžete prohlédnout návrh doplněný o zvolené technologie.



Obrázek 2.4: Detailní pohled na backend

Kapitola 3

Implementace manažera stavu

V této kapitole popisuji implementaci manažera stavu a některé problémy, které jsem při ní řešil. Výsledný kód naleznete na přiloženém CD v adresáři „StateManager“.

3.1 Napojení na zbytek systému

Způsob napojení manažera stavu na vyšší vrstvy systému je definován v požadavku R3. Během implementace jsem došel k závěru, že napojení na Salt budu řešit stejným způsobem. Kromě endpointů určených pro vyšší vrstvy tedy API obsahuje i endpointy, které využívá výhradně konfigurační server. Ty umožňují získat téměř stejná, pouze jinak strukturovaná data. Kompletní dokumentaci API naleznete v příloze A.

Kromě REST rozhraní interaguje manažer stavů se Saltem i přímo. Po každé změně stavu zavolá program „salt“, který si z API stáhne aktuální data a zajistí jejich aplikaci na výpočetní nody. Toto napojení funguje díky idempotenci běhů saltu. Je možné ho spustit mnohokrát a výsledkem bude vždy stejný stav, odpovídající aktuálním datům v databázi. Více o této problematice najdete v sekci 4.6.

3.2 Struktura programu

Manažer stavu je strukturován podle návrhového vzoru model-view-presenter.

Jako modely slouží třídy ORM systému Mongoose doplněné o metody implementující aplikační logiku. Pro Mongoose jsem sáhl v průběhu implementace, když začalo být jasné, že stávající modely obsahují velké množství boilerplate kódu. Usnadňuje práci s databází a dodává modelům jasnou strukturu.

Na místě pohledů slouží framework Hapi a rozsáhlé validátory (Joi [36]), kterými prochází data při cestě z i do aplikace. Vstupní validace slouží k sa-

nitaci vstupů a generování uživatelsky přívětivých chybových zpráv. Výstupní validace mi umožňuje rychle najít chyby v programu a zabraňuje nežádoucím efektům, které by mohly nastat, kdyby se vadná data dostala do zbytku systému. Pokud výstupní validace detekuje chybu, program odpoví na dotaz HTTP kódem 500 (internal server error) a žádná data nevrátí.

Modely a pohledy spojují presentery. Obsahují funkce volané frameworkem Hapi poté, co požadavek projde vstupní validací. Presentery následně podle druhu dotazu zavolají odpovídající model a provedou s ním požadované operace.

3.3 Řešené problémy

Během implementace jsem narazil na několik problémů. Předkládám je zde spolu s popisy mých řešení.

3.3.1 Asynchronní I/O

Kód psaný pro Node.JS typicky běží v jediném vlákně. To významně zjednodušuje tvorbu programů, protože programátor nemusí pamatovat na ošetřování operací zámky ani řešit problémy vzniklé chybnou interakcí mezi různými vlákny. Na první pohled by se mohlo zdát, že spolu s vlákny se musíme vzdát paralelního zpracování a tím i velké části výkonu na moderních počítačích. Ve skutečnosti to ale není pravda, protože popsaná synchronicita se netýká I/O operací. Moduly jako mongo (ovladač databáze) a fs (přístup k filesystému) totiž spouštějí kód, který neblokuje běh hlavního programu. Zpracovává se na pozadí za použití pracovních vláken, které spravuje Node.JS. Ve chvíli, kdy je dříve spuštěná operace dokončena, je notifikováno hlavní vlákno programu a spuštěn „callback“. Tak je označována funkce předaná za tímto účelem při startu asynchronní operace.

Na příkladech 3.1 a 3.2 si můžete prohlédnout dvě varianty stejného kódu. První varianta předpokládá, že volané funkce jsou synchronní (zablokují program do svého ukončení). Druhá varianta ukazuje, jak by kód vypadal, pokud by funkce byly asynchronní. Na první pohled je zřejmé, že druhá varianta je méně přehledná. Čitelnost se navíc s rostoucí složitostí kódu exponenciálně zhoršuje a může vést až k takzvanému „callback hell“. Tímto termínem je označován absolutně neudržovatelný kód obsahující velké množství do sebe vnořených callbacků.

Příklad 3.1: řetězení synchronních funkcí

```
var x = getDataFromFile();
var y = getNextData(x);
var result = getAnotherData(y+'25');
console.log(result);
```

Příklad 3.2: řetězení asynchronních funkcí

```
getDataFromFile(function(x) {
  getNextData(x, function(y) {
    getAnotherData(y+'25', function(res) {
      console.log(res);
    });
  });
});
```

Abych se „callback hellu“ vyhnul, používám návrhový vzor Promise[37]. Asynchronní funkce podle tohoto vzoru nejprve synchronně vrátí zástupný objekt (promise), do kterého později doplní výsledek operace. S oním zástupným objektem lze do té doby nakládat. Například ho předat jiné funkci nebo řetězit. Třída Promise je bohužel přítomna až ve standardu EcmaScript 6. Ten ještě není implementován ve stabilní verzi Node.JS a tak používám modul es6-shim [38] pro její doplnění za běhu. Příklad 3.3 ukazuje jednoduché zřetězení funkcí za použití vzoru Promise. Všimněte si zejména nahrazení čtyř úrovní zanoření pouze dvěma. Tento script, doplněný o implementaci funkcí, najdete také na přiloženém CD v adresáři „promise_example“.

Příklad 3.3: řetězení asynchronních funkcí pomocí Promise

```
getDataFromFile()
.then(getNextData)
.then(function(y) {
  return getAnotherData(y+'25');
})
.then(console.log);
```

3.3.2 Unikátní indexy v MongoDB

Pro korektní běh manažera stavů bezpodmínečně vyžaduji, aby databáze zaručila unikátnost některých polí v rámci celé databáze (mac adresa) nebo v rámci skupiny hypervizorů (hgroupseq). Pokud se aplikace pokusí zapsat data, která požadavek na unikátnost porušují, požadavek musí být odmítnut. To se může výjimečně stát u náhodně generovaných polí jako je mac adresa. Častěji se to ale objeví u sekvenčních čísel hgroupseq. V případě, že dva klienti vyžádají vytvoření záznamu ve stejný okamžik, pak jeden z dotazů na databázi selže. Program situaci podle chybového kódu databáze vyhodnotí jako kolizi a po přegenerování identifikátoru dotaz opakuje.

Bohužel implementace unikátních indexů v MongoDB není dokonalá. V průběhu vývoje jsem narazil na dvě její omezení. Prvním z nich je, že kontrola indexu se nevztahuje na poddokumenty v rámci jednoho dokumentu. Uvažujme data z příkladu 3.4 a dále, že v databázi je definován unikátní index nad polem nics.mac. Pokus o přidání nové síťové karty s mac „00:a7:46:6f:2f:d5“ do VM test1 podle očekávání selže. Přidání tohoto záznamu do VM test2 ale

databáze povolí a způsobí tím nekonzistenci dat v systému (mac již nebude unikátní, VM test2 bude obsahovat dvě síťové karty se stejnou mac). Tento problém lze naštěstí lehce obejít kontrolou indexovaných polí před každým zápisem do databáze. Atomicita zápisu v rámci jednoho dokumentu v tomto případě zaručí, že kolize dvou různých pokusů o editaci neskončí nekonzistencí v systému.

Příklad 3.4: zjednodušený export části databáze virtuálních strojů

```
[
  {
    "_id": ObjectId("552fd7382475a7f937ebfb19"),
    "name": "test1",
    "hgroup" : ObjectId("5467537f79cad114163ee769"),
    "hddidxhelper" : ObjectId("5467537f79cad114163ee769"),
    "hdds" : [
      {
        "hgroupseq" : 2,
        "storage" : "local ZFS",
        "size" : 5,
      },
      {
        "hgroupseq" : 4,
        "storage" : "local ZFS",
        "size" : 50,
      }
    ],
    "nics" : [
      {
        "mac" : "0a:d1:46:ef:7f:e5",
      }
    ]
  },
  {
    "_id": ObjectId("552fad552475a7f937ebfb15"),
    "name": "test2",
    "hgroup" : ObjectId("5467537f79cad114163ee769"),
    "hddidxhelper" : ObjectId("552fad552475a7f937ebfb15"),
    "hdds" : [ ],
    "nics" : [
      {
        "mac" : "00:a7:46:6f:2f:d5",
      }
    ]
  }
]
```

Druhý, z mého pohledu závažnější, nedostatek unikátních indexů se projeví ve chvíli, kdy definuji index složený z více polí, z nichž některá se v dokumentu nemusí vyskytovat (compound sparse unique index). Tento typ indexu používám u pole hdds.hgroupseq, které reprezentuje pořadové číslo disku v rámci

skupiny výpočetních nodů. Pokud je index definován jako sparse, pak do něj MongoDB nepřidá prvky null a těch tak může být v databázi libovolné množství. Složený index ale tuto podmínku splní jen v případě, že všechny jeho prvky jsou null. Problém tak nastane ve chvíli, kdy se program pokusí do databáze uložit dva virtuální stroje bez disků náležící do stejné skupiny. První z nich totiž obsadí v indexu pozici „[hgroupid, null]“ a pokus o přidání druhého takového VM selže.

Tuto nepříjemnou situaci řeším pomocí pomocného pole hddidxhelper v databázi. Těsně před zapsáním záznamu nastavím toto pole na stejnou hodnotu jakou má id VM, pokud ten nemá žádné disky. Pokud disky obsahuje, pak toto pole nastavím na stejnou hodnotu jakou má id skupiny (hgroup). Do unikátního indexu pak zahrnu i pole hddidxhelper. Toto opatření zajistí splnění unikátnost záznamu i v případě, že VM nemá disky (trojice [hgroupid,vmid,null] je vždy unikátní) a také správnou funkci indexu v opačném případě (zdvojení jedné z hodnot index neovlivní). Nevýhoda tohoto řešení je závislost na kusu aplikační logiky, do kterého mohu lehce zanést chybu.

Podpora pro filtrování indexů přímo v databázi by tento problém úplně eliminovala a je v plánu pro MongoDB 3.1. Release této verze se očekává do konce roku 2015. [39]

3.3.3 Reprezentace změn VM

V průběhu změny parametrů virtuálního stroje musím v databázi držet data o starém i novém stavu. Abych nemusel ukládat všechna data dvakrát a pro usnadnění případné budoucí podpory pro částečné updaty (HTTP metoda PATCH) jsem se rozhodl reprezentovat nový stav pouze jako seznam změn. Tento seznam formátuji podle standardu rfc6902 [40] za pomoci modulu rfc6902-json-diff[41].

Drobný problém s touto reprezentací nastal během přechodu na mongoose modely. Modul jsonpatch-js[42], který používám k získání nového stavu VM z diffu a starého stavu, totiž používá javascriptový operátor delete. Na tom samo o sobě není nic špatného. Problém dělá kompatibilita s mongoose, který má s delete potíže (delete maže accessory místo dat). Vytvořil jsem tedy fork[43] projektu json-patch-js a upravil ho tak, aby se v problémových případech operátoru delete vyhnul.

3.3.4 Správa obrazů CD a templatů

Požadavek R11 vyžaduje podporu pro NFS úložiště obsahující obrazy CD a disků. Návrh ani zprovoznění tohoto úložiště ale nejsou součástí této práce a úložiště ani jeho specifikace nebylo v průběhu implementace dostupné. Z tohoto důvodu jsem implementoval vlastní dočasné řešení pro ukládání a distribuci obrazů CD a disků. Toto řešení bude z backendu odstraněno po nasazení centrálního NFS úložiště, jak ho specifikují požadavky.

Dočasné řešení spočívá ve vytvoření kolekce cdimages a templates v databázi a v uložení souborů v adresářové struktuře Saltu. Salt tyto soubory pak automaticky distribuuje výpočetním nodům. Tento systém distribuce vyžaduje dostatek úložného prostoru pro všechny obrazy na všech výpočetních nodech a není tak vhodný do prostředí, kde by jich bylo větší množství. Pro testování a ověření funkčnosti několika obrazů je ale dostatečný.

3.4 Testování

Pro testování manažera stavu používám nástroj Lab[44]. Během vývoje jsem vytvořil 43 testů pokrývajících 92% kódu (test report najdete na CD). Většina z nich se soustředí na interakci s API. Ostatní testují samostatné modely. V současné době bohužel nemám žádné integrační testy, které by ověřovaly správné napojení zbytku systému na manažera stavů.

Kapitola 4

Konfigurace a integrace Saltu

Druhou nejdůležitější součástí backendu je konfigurační systém Salt. V následujících sekcích popisuji jeho napojení na manažera stavu a způsob jakým konfiguruje výpočetní nody. Veškeré konfigurační soubory a scripty Saltu najdete na CD v adresáři „SaltMaster“.

4.1 Napojení na manažer stavu

Pro komunikaci s manažerem používám dvě metody. První z nich je takzvaný externí pillar. Jedná se o krátký python script, který se při každém spuštění Saltu připojí na API a stáhne z něj aktuální data. Tato data poté integruje do pillaru, což je struktura dostupná během konfigurace všem modulům a stavům v Saltu.

Druhou metodu používám pro zasílání dat zpět na API. Informace o aktuálních běžících VM nebo zprávy o průběhu konfigurace nejprve po Salt komunikační sběrnici posílám na master. Tam sběrnici poslouchá python script a filtruje zprávy týkající se manažera stavů. Tyto zprávy ze sběrnice vyzvedne a jako samostatné požadavky pošle na API.

Obě metody využívají integrovaných funkcí Saltu a na výpočetních nodech nevyžadují žádnou konfiguraci. Vše je centralizováno na masteru.

4.2 Salt stavy

Popisu konfigurace cílového serveru se v Salt terminologii říká stav (state). Většinou se nacházejí v souborech s příponou sls, kde jsou zapsané v YAML formátu. To ale není jediný způsob jak konfiguraci tvořit. Salt je v tomto ohledu mimořádně flexibilní. Například při automatické distribuci obrazů CD a disků používám stavový soubor reprezentovaný python scriptem. Ten na konci vrátí datovou strukturu, kterou salt zpracuje namísto YAML souboru.

I běžné YAML stavy mohou obsahovat dynamicky generované části. O to se stará jinja2 templatovací systém, který je před každým během Saltu nad sls soubory spuštěn. Umožňuje významně upravit výsledek třeba na základě dat z pillaru (ve kterém jsou i aktuální data z API). Tímto způsobem tvořím i konfiguraci virtuálních strojů. Na příkladu 4.1 si můžete prohlédnout jeden Salt stav. Jedná se o část sls souboru starajícího se o konfiguraci KVM virtuálních strojů. Všimněte si zejména jinja templatování uvozeného složenými závorkami.

Příklad 4.1: stav zajišťující konfiguraci VM na Qemu

```
# ...
{% for vm in pillar.get('vms', {}) %}
# ...
{{vm.id}}-realstate:
  bgqemu.present:
    - name: {{vm.id}}
    - vmstate: {{vm.state}}
    - cpu_count: {{vm.cpuno}}
    - ram_size: {{vm.ram}}
    {% if vm.cdrom is defined %}
    - iso_path: /bigcloud/files/{{vm.cdrom}}.iso
    {% endif %}
    - vnc_display: {{vm.VNCPort - 5900}}
    - vnc_password: {{vm.VNCPassword}}
    - disks:
{%- for disk in vm.hdds %}
      - {path: {{devpath(disk)}}}
{%- endfor %}
    - nics:
{%- for vif in vm.nics %}
      - vifname: tap{{vm.hgroupseq}}.{{loop.index}}
        ip: {{vif.ip}}
        bridge: "vmbr"
        vlan: {{vif.vlan}}
        mac: "{{vif.mac}}"
{%- endfor %}
    - nicupscript: /bigcloud/ifup
    - nicdownscript: /bigcloud/ifdown
    - require:
      - cmd: {{vm.id}}-state
  {% if vm.cdrom is defined %}
      - file: {{vm.cdrom}}.iso-getfile
  {% endif %}

# ...
{% endfor %}
# ...
```

4.3 Podpora pro LVM a Xen

Základní podpora pro LVM a Xen v tomto systému byla mou první zkušeností se Saltem. Ani pro jeden z nich není v Saltu připraven modul, který by splňoval mé požadavky. Nutnou funkcionalitu jsem tedy doplnil pomocí shellových scriptů. To se zejména u LVM neukázalo jako nejlepší řešení. Kód pro vytváření, mazání, změnu a čištění lvm oddílů během vývoje narostl a díky omezením (návratové hodnoty, práce se strukturovanými daty) shellu je jeho dekompozice na menší funkce velmi složitá. I přes to v současné době vše v pořádku funguje. V případě potřeby přidání nových funkcí ale tyto scripty opustím a napíšu pro správu Xenu a LVM pythonové moduly, jako jsem to udělal pro ZFS a Qemu.

4.4 Podpora pro ZFS a Qemu

Pracovat se ZFS a Qemu jsem začal již poučen problémy s shellovými scripty. Rovnou jsem tedy napsal kód v pythonu. To Salt značně usnadňuje, protože umožňuje vytváření uživatelských modulů, které automaticky distribuuje na všechny servery. Následně je možné je volat z sls souborů.

Oproti Xenu byla integrace Qemu náročnější. Částečně to způsobila nekompletní dokumentace a částečně absence alternativy libxl (a programu xl) pro Qemu. Libvirt, který je za tímto účelem často používán, jsem zavrhl pro přílišnou složitost. Ve výsledku komunikuji s běžícími instancemi Qemu přímo pomocí protokolu QMP a žádného prostředníka tak nevyžaduji.

4.5 Podpora pro OpenVSwitch

Salt OpenVSwitch přímo nekonfiguruje. Pouze zajišťuje přítomnost scriptů, které Xen i Qemu volají při vytváření virtuálních síťových rozhraní. Tyto scripty zajišťují i přidání bezpečnostních pravidel do flow tabulek OpenVSwitche. Tato pravidla umožňují komunikovat virtuálnímu stroji pouze s IP a MAC adresou, která mu byla přidělena. Navíc znemožňují provozování falešných DHCP serverů. Na příkladu 4.2 si je můžete prohlédnout.

Příklad 4.2: flow pravidla přidávaná k portům virtuálních strojů

```
priority=10011,in_port=$port,udp,tp_dst=68,action=drop
priority=10010,in_port=$port,dl_src=$mac,ip,\
  nw_src=$ip,action=normal
priority=10010,in_port=$port,dl_src=$mac,arp,\
  nw_src=$ip,action=normal
priority=10000,in_port=$port,action=drop
```

4.6 Idempotence operací

Během normálního fungování backendu jsou stavy běžně aplikovány vícekrát. Každý běh saltu kontroluje a v případě potřeby aplikuje všechny stavy na daném stroji. Je tedy důležité, aby byly stavy idempotentní. Většina tuto vlastnost přirozeně má. Jsou ale výjimky, jako je aplikování template na disk. Je nepřípustné, aby tato akce proběhla několikrát. Došlo by ke ztrátě dat, která se na disku od posledního běhu objevila.

V případě takto kritických operací využívám pomocný soubor. Do něj si zapisuji úspěšné dokončení operace a její id. Před začátkem operace soubor zkontroluji, a pokud již id operace obsahuje, tak jí přeskočím. Byla již provedena dříve.

Kapitola 5

Nasazení do testovacího prostředí

Tato kapitola se věnuje návrhu testovacího prostředí a nasazení backendu. Výsledkem je funkční prototyp připravený přijímat požadavky a vytvářet virtuální stroje.

5.1 Dostupné prostředky

Pro nasazení mám k dispozici následující prostředky:

- 7x 1U server SuperMicro s procesorem Intel Core i3-4150, 16GB RAM, 2x 2TB diskem, 2x 1Gb/s síťovou kartou a out of band managementem
- 4x 10Gb/s síťová karta Intel 82599ES s dvěma SFP+ sloty
- 1x switch Netgear GS728TXS (24 1Gb/s portů, 4 10Gb/s porty)
- 2x switch Netgear bez možnosti konfigurace (24 1Gb/s portů)
- 1x RouterBoard RB750
- 1x vývojový virtuální server dostupný po Internetu

5.2 Boot ze sítě a NFS root

Protože výpočetních nodů může být v systému velké množství, je rozumné co možná nejvíce zjednodušit jejich správu a údržbu. Připravím pro ně DHCP, TFTP a network file system (NFS) server za účelem automatizace jejich instalace. Všechny výpočetní nody budou bootovat ze sítě a root filesystému budou mít také připojený po síti. Nový server tak bude možné pouze připojit a zanést do systému. Instalace operačního systému nebude nutná. Eliminuji tím i systémové disky, které by jinak musely být v každém nodu.

Konfigurační soubory všech služeb a nástrojů popsaných v této sekci naleznete na přiloženém CD v adresáři bootserver.

5.2.1 DHCP

Pro boot ze sítě je vyžadován DHCP server. Přidělí klientům IP adresu a také je odkáže na TFTP server, ze kterého si stáhnou bootloader. Navíc je možné klientům předat další parametry, sloužící k jejich konfiguraci. Já předávám informace o hostname a typu hypervizoru, které využívám dále při bootování (viz sekce 5.2.3).

Jako DHCP server používám klasickou implementaci od Internet Systems Consorcia. [45]

5.2.2 TFTP

Přenos souborů během bootu ze sítě zajišťuje TFTP server. TFTP funguje na podobném principu jako FTP, ale oproti němu je mnohem jednodušší. Nemá žádnou podporu pro autentizaci nebo šifrování, dokonce ani pro výpis obsahu adresáře.

Jako TFTP server používám software tftp-hpa [46].

5.2.3 Bootloader

Bootloader je první program, který se začne na serveru spouštět. Jeho úkolem je získat, načíst a spustit jádro operačního systému. V mém případě je bootloader načten z TFTP serveru a z něj také musí získat jádro. Protože prostředí nebude uniformní (více různých jader a jejich parametrů), zvolil jsem bootloader Grub2. Jeho konfigurační soubor má totiž formu scriptu, ve kterém lze používat podmíněné výrazy a dokonce i smyčky[47]. Na základě parametrů předaných DHCP serverem se tak přímo během bootování může rozhodnout jaké jádro načíst a jaké parametry mu předat.

5.2.4 NFS

Root filesystému je podle instrukcí předaných bootloaderem (parametr jádra nfsroot) připojen z NFS serveru. NFS je v Linuxu integrovaný systém pro sdílení souborů po síti a umožňuje použití i jako jediné úložiště v systému.

5.2.5 Automatická instalace OS

V centrálním NFS úložišti má každý výpočetní node svůj adresář s obsahem systémového disku. Tento obsah automaticky generuji pomocí programu multistrap. Na rozdíl od klasického Debian debootstrapu, umožňuje detailnější upravení instalačního procesu včetně využití více než jednoho repozitáře. [48]

5.3 Návrh zapojení

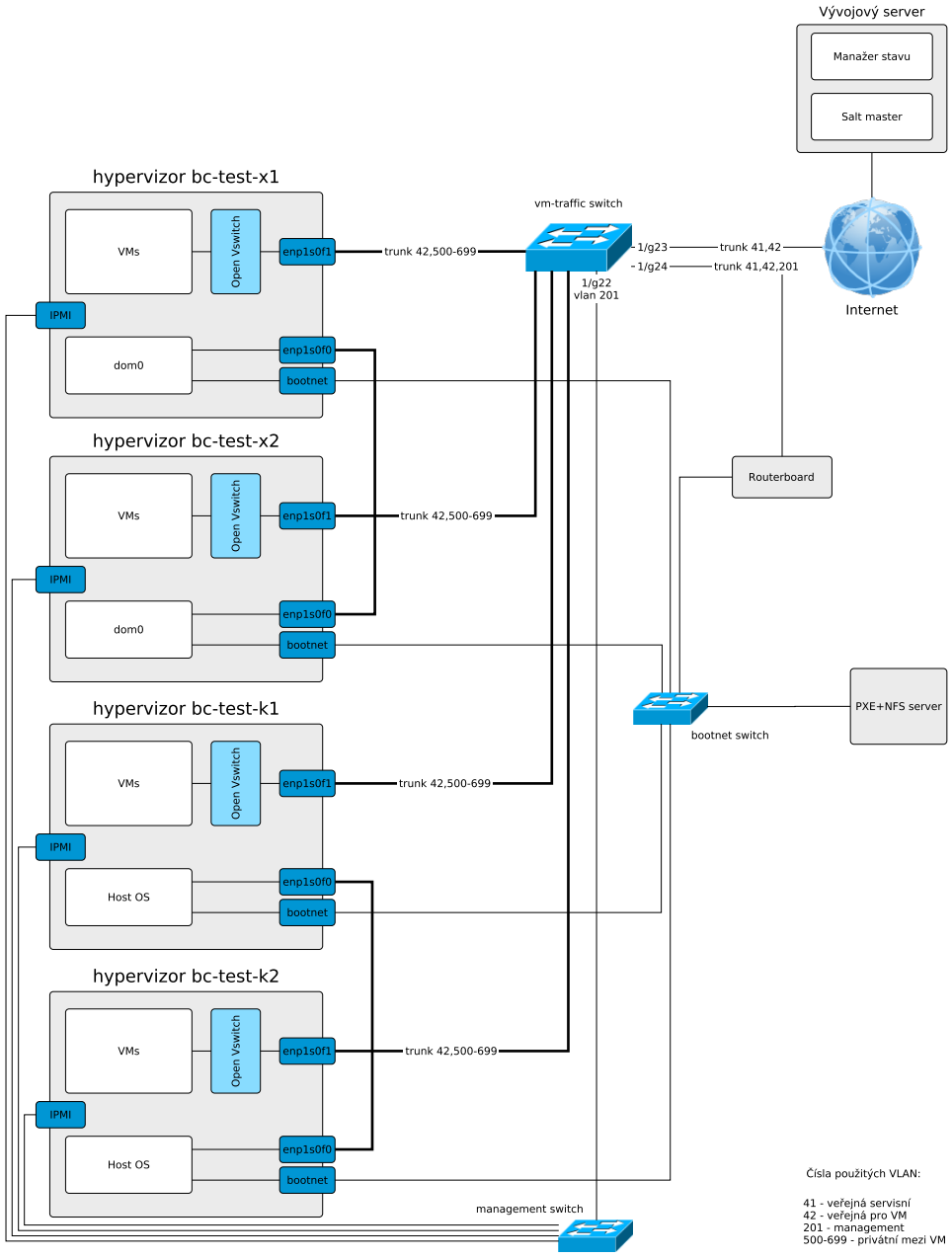
Testovací prostředí se musí skládat alespoň ze dvou skupin výpočetních nodů. V jedné skupině poběží hypervizory Xen zatímco ve druhé hypervizory KVM. V minimální konfiguraci tedy potřebuji pro výpočetní nody 4 servery.

Přenosy velkého množství dat očekávám v síti na dvou místech. Jednak na linkách využívaných jednotlivými virtuálními stroji a poté na propojích používaných pro sdílené datové úložiště. V této fázi projektu sice ještě není implementované, ale testovací prostředí na něj musí být připravené. Síť pro VM musí být společná pro všechny výpočetní nody, k jejímu vytvoření tedy využiju čtyři 10Gb/s porty, které mám k dispozici na switchi. Síti pro sdílená úložiště stačí, když bude dostupná pouze v rámci skupiny nodů. Využiju pro ní zbývající SFP porty na serverech a propojím je přímo. Dále budu potřebovat oddělené sítě pro management a pro bootování. Na těchto sítích ale nebude velký provoz a vystačí si tak s obyčejnými 1Gb/s switchi.

Některé části sítě by neměly být veřejně dostupné. Navzdory tomu je ale žádoucí, aby stroje v nich umístěné, měly přístup k Internetu. To umožní router, který pomocí firewallu a překladu adres zajistí propuštění pouze vyžádaných dat. Router budu také využívat pro vzdálený přístup do testovacího prostředí pomocí integrovaného VPN serveru.

Pro oddělení různých sítí budu používat VLANy. Potřebuji minimálně dvě pro veřejné sítě, jednu pro management (IPMI, switche) a určité množství rezervované pro privátní sítě mezi VM. První veřejná VLAN slouží pro vzdálený management systému. Druhá pro provoz virtuálních strojů. Rezervace a konfigurace privátních VLAN je nutná, protože switch neznámé VLAN nepropustí.

Diagram výsledného testovacího prostředí si můžete prohlédnout na obrázku 5.1.



Obrázek 5.1: Testovací prostředí

Závěr

Navržená architektura se během implementace a nasazení ukázala jako vhodná. Nenarazil jsem na žádné zásadní problémy a využití existujících technologií spolu s filosofií „méně je někdy více“ mi umožnilo včas dokončit poměrně rozsáhlý systém. V současné době backend v testovacím provozu zajišťuje vytváření a správu virtuálních strojů na hypervizorech Xen a KVM. Podporuje ukládání virtuálních disků na ZFS a LVM oddíly a umožňuje připojení VM k sítím, které zabezpečuje proti zneužití. Využití všech jeho funkcí je možné pomocí zdokumentovaného HTTP REST API.

Dalším krokem ve vývoji je probíhající důkladné testování ve spolupráci se zadavatelem. Po něm bude následovat implementace dalších funkcí, z nichž nejdůležitější je integrace DHCP serveru a podpora pro sdílená datová úložiště.

Osobně si z projektu odnáším zkušenosti s MongoDB, programováním pro Node.JS, správou serverů a sítí a také s integrací komplexního systému.

Cílem této práce bylo navrhnout backend cloudového systému a následně implementovat a nasadit jeho prototyp do testovacího prostředí. Všechny části zadání i požadavky zadavatele určené pro prototyp byly splněny.

Slovník

API application programming interface. Rozhraní pro přístup k programu, použitelné jiným programem.

BSON distributed replicated block device. Binární reprezentace formátu JSON rozšířená o více datových typů. [49].

COW copy on write.

DHCP dynamic host configuration protocol.

DRBD distributed replicated block device. „Blokové zařízení navžené jako stavební blok vysoce dostupných clusterů“ [50].

FTP file transfer protocol.

HA high availability. Vysoká dostupnost. Charakteristika systému schopného fungovat navzdory neočekávanému selhání nebo chybě.

HTTP hypertext transfer protocol.

I/O input/output.

Jinja2 „Moderní a pro designery přívětivý templatovací jazyk pro Python“[51].

KVM kernel virtual machine.

LVM2 logical volume manager v2.

NFS network file system.

ORM Object-relational mapping.

QMP qemu management protocol.

RAID redundant array of independant disks. Spojení více fyzických disků za účelem ochrany dat a/nebo zvýšení rychlosti datového úložiště.

RAM random access memory.

REST representational state transfer.

RSPAN remote switched port analyzer.

SQL structured query language.

TFTP trivial file transfer protocol.

VM virtual machine. Virtuální stroj. Virtuální počítač. Izolované výpočetní prostředí vytvořené hypervizorem.

VNC virtual network computing. Sytém pro vzdálené sledování grafického výstupu počítače a pro interakci s tímto výstupem pomocí přeposílání stisků kláves a pozice myši.

YAML YAML Ain't Markup Language (rekurzivní akronym). Standard pro datovou serializaci dobře čitelný pro lidi.[52].

ZVOL ZFS volume.

Literatura

[1] Gartner, Inc.: Cloud Computing - Gartner IT glossary. [Cited 2015-02-01]. Dostupné z: http://www.gartner.com/it-glossary/cloud-computing/

[2] Citrix Systems, Inc.: XenServer Tech Info. [Cited 2015-02-20]. Dostupné z: http://www.citrix.com/products/xenserver/tech-info.html

[3] Russell Pavlicek, Lars Kurth: XAPI. [Cited 2015-02-20]. Dostupné z: http://wiki.xenproject.org/wiki/XAPI

[4] OpenStack Foundation: OpenStack Open Source Cloud Computing Software. [Cited 2015-02-25]. Dostupné z: http://www.openstack.org/

[5] OpenStack Foundation: Get started with OpenStack. [Cited 2015-02-25]. Dostupné z: http://docs.openstack.org/admin-guide-cloud/content/ch_getting-started-with-openstack.html

[6] Ignacio M. Llorente: Eucalyptus, CloudStack, OpenStack and OpenNebula: A Tale of Two Cloud Models. [Cited 2015-02-25]. Dostupné z: http://opennebula.org/eucalyptus-cloudstack-openstack-and-opennebula-a-tale-of-two-cloud-models/

[7] OpenNebula Project: OPENNEBULA KEY FEATURES. [Cited 2015-02-25]. Dostupné z: http://opennebula.org/about/key-features/

[8] Linkedin corp.: Rest.li. [Cited 2015-02-26]. Dostupné z: http://rest.li/

[9] INTRIDEA Inc.: Grape | REST-like API micro-framework. [Cited 2015-02-26]. Dostupné z: http://intridea.github.io/grape/

[10] Kris Jordan: Recess PHP Framework. [Cited 2015-02-26]. Dostupné z: http://www.recessframework.org/

[11] více autorů: A rich framework for building applications and services. [Cited 2015-02-26]. Dostupné z: http://hapijs.com/

[12] Joyent, Inc: Node.js. [Cited 2015-02-26]. Dostupné z: https://nodejs.org/

[13] Ecma International: Standard ECMA-262. [Cited 2015-02-26]. Dostupné z: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[14] solidIT consulting & software development gmbh: DB-Engines Ranking. [Cited 2015-03-11]. Dostupné z: http://db-engines.com/en/ranking

[15] Oleg Efimov: Node-mysql-libmysqlclient. [Cited 2015-03-11]. Dostupné z: http://sannis.github.io/node-mysql-libmysqlclient/

[16] Brian Carlson: node-postgres. [Cited 2015-03-11]. Dostupné z: https://github.com/brianc/node-postgres

[17] MongoDB, Inc.: Node.js MongoDB Driver. [Cited 2015-03-11]. Dostupné z: http://docs.mongodb.org/ecosystem/drivers/node-js/

[18] Oracle Corporation: MySQL HA/Scalability Guide. [Cited 2015-03-11]. Dostupné z: http://dev.mysql.com/doc/mysql-ha-scalability/en/ha-overview.html

[19] Oracle Corporation: MySQL Cluster Installation. [Cited 2015-03-11]. Dostupné z: http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster-installation.html

[20] Oracle Corporation: Overview of MySQL with DRBD/Pacemaker/-Corosync/Oracle Linux. [Cited 2015-03-11]. Dostupné z: http://dev.mysql.com/doc/mysql-ha-scalability/en/ha-drbd.html

[21] The PostgreSQL Global Development Group: High Availability, Load Balancing, and Replication. [Cited 2015-03-11]. Dostupné z: http://www.postgresql.org/docs/9.4/static/high-availability.html

[22] MongoDB, Inc.: FAQ: MongoDB Fundamentals. [Cited 2015-03-15]. Dostupné z: http://docs.mongodb.org/manual/faq/fundamentals/

[23] MongoDB, Inc.: FAQ: Replication and Replica Sets. [Cited 2015-03-15]. Dostupné z: http://docs.mongodb.org/manual/faq/replica-sets/

[24] Ansible, Inc.: Ansible Documentation: About Ansible. [Cited 2015-03-17]. Dostupné z: http://docs.ansible.com/

[25] Ansible, Inc.: Ansible Documentation: Intro to Playbooks. [Cited 2015-03-17]. Dostupné z: http://docs.ansible.com/playbooks_intro.html

[26] SaltStack Inc.: Introduction to Salt. [Cited 2015-03-25]. Dostupné z: http://docs.saltstack.com/en/latest/topics/index.html

[27] SaltStack Inc.: Salt SSH. [Cited 2015-03-25]. Dostupné z: http://docs.saltstack.com/en/latest/topics/ssh/index.html

[28] SaltStack Inc.: Reactor System. [Cited 2015-03-25]. Dostupné z: http://docs.saltstack.com/en/latest/topics/reactor/index.html

[29] Chef Software, Inc.: An Overview of Chef. [Cited 2015-04-21]. Dostupné z: https://docs.chef.io/chef_overview.html

[30] Puppet Labs: Puppet 4.0 Docs: Overview of Puppet's Architecture. [Cited 2015-04-21]. Dostupné z: https://docs.puppetlabs.com/puppet/4.0/reference/architecture.html

[31] SaltStack Inc.: Reactor System. [Cited 2015-03-25]. Dostupné z: http://docs.saltstack.com/en/latest/topics/virt/

[32] Open vSwitch: Production Quality, Multilayer Open Virtual Switch. [Cited 2015-04-21]. Dostupné z: http://openvswitch.org/

[33] Thomas Graf: Why Open vSwitch? [Cited 2015-04-21]. Dostupné z: https://github.com/openvswitch/ovs/blob/master/WHY-OVS.md

[34] Xen Project: Xen Project Software Overview. [Cited 2015-04-22]. Dostupné z: http://wiki.xenproject.org/wiki/Xen_Overview

[35] Fabrice Bellard: KVM. [Cited 2015-04-22]. Dostupné z: http://wiki.qemu.org/KVM

[36] Nicolas Morel: hapijs/joi - Object schema validation. [Cited 2015-04-29]. Dostupné z: https://github.com/hapijs/joi

[37] Promises/A+ organization: Promises/A+. [Cited 2015-04-29]. Dostupné z: https://promisesaplus.com/

[38] Paul Miller: es6-shim. [Cited 2015-04-29]. Dostupné z: https://github.com/paulmillr/es6-shim/

[39] Scott Hernandez: Support Filtered (Partial) Indexes. [Cited 2015-04-29]. Dostupné z: https://jira.mongodb.org/browse/SERVER-785

[40] P. Bryan, Ed.: JavaScript Object Notation (JSON) Patch. [Cited 2015-04-29]. Dostupné z: https://tools.ietf.org/html/rfc6902

[41] Marten Lienen: JSON diff according to RFC6902. [Cited 2015-04-29]. Dostupné z: https://github.com/CQQL/rfc6902-json-diff-js

[42] Byron Ruth: jsonpatch-js. [Cited 2015-04-29]. Dostupné z: https://github.com/bruth/jsonpatch-js

[43] Byron Ruth: jsonpatch-js. [Cited 2015-04-29]. Dostupné z: https://github.com/Gregy/jsonpatch-js

[44] Wyatt Preul: lab: Node test utility. [Cited 2015-04-29]. Dostupné z: https://github.com/hapijs/lab

[45] Internet Systems Consortium, Inc.: ISC DHCP: Enterprise Grade Solution for Configuration Needs. [Cited 2015-04-22]. Dostupné z: https://www.isc.org/downloads/dhcp/

[46] H. Peter Anvin: tftp-hpa. [Cited 2015-04-22]. Dostupné z: https://www.kernel.org/pub/software/network/tftp/

[47] Free Software Foundation, Inc.: GNU GRUB Manual 2.00. [Cited 2015-04-22]. Dostupné z: http://www.gnu.org/software/grub/manual/grub.html

[48] Neil Williams: Multistrap. [Cited 2015-04-22]. Dostupné z: https://wiki.debian.org/Multistrap

[49] MongoDB, Inc.: JSON AND BSON. [Cited 2015-03-15]. Dostupné z: http://www.mongodb.com/json-and-bson

[50] LINBIT HA-Solutions GmbH: DRBD: What is DRBD. [Cited 2015-03-11]. Dostupné z: http://drbd.linbit.com/

[51] Armin Ronacher: Welcome to Jinja2. [Cited 2015-04-21]. Dostupné z: http://jinja.pocoo.org/docs/dev/

[52] Clark C. Evans: The Official YAML Web Site. [Cited 2015-03-17]. Dostupné z: http://yaml.org/

Příloha A

Dokumentace API

BigCloud backend API umožňuje spravovat (CRUD) virtuální stroje a získávat informace o dostupné infrastruktuře backendu. API dodržuje zavedené praktiky HTTP REST rozhraní jako bezstavovost a zachovávání významu HTTP metod.

A.1 Komunikace

Ve výchozím nastavení naslouchá manažer stavů na 127.0.0.1:3000. To lze změnit v konfiguračním souboru (config/main.json).

Komunikace s API probíhá výhradně pomocí HTTP předáváním JSON dokumentů. Šablona URL pro přístup k API vypadá takto:

```
http://127.0.0.1:3000/v<verze>/<zdroj>[/<id>[/<podzroj>]]
```

Znaky „[“ a „]“ uvozují nepovinné části. Znaky „<“ a „>“ uvozují parametry. Za parametr **verze** klient dosadí číslo verze API, se kterým si přeje komunikovat. V současné době je implementována pouze verze 1. Parametr **zdroj** může nabývat několika různých hodnot, seznam následuje dále v dokumentaci. Parametr **id** identifikuje specifickou instanci zdroje, se kterým chce klient pracovat. Id jsou typicky alfanumerické řetězce o délce 24 znaků. Některé zdroje mají **podzdroje**, ke kterým lze přistoupit pomocí posledního parametru. Více o nich dále v dokumentaci.

HTTP operace využívané v API jsou GET (čtení dat), POST (vytváření nových objektů), PUT (editace stávajících objektů) a DELETE (mazání stávajících objektů).

A.2 Návratové kódy

API odpoví na každý přijatý dotaz. Úspěšnost operace lze odvodit z HTTP kódu:

200 Operace proběhla úspěšně.

201 Požadavek byl zpracován a na jeho základě byl vytvořen nový objekt. URL nového objektu je předáno v hlavičce Location.

202 Požadavek byl přijat a předán ke zpracování konfiguračnímu serveru. Operace ale nemusí nutně uspět.

400 Požadavek neodpovídá validačním pravidlům nebo obsahuje odkaz na neexistující objekt.

404 Objekt se nepodařilo najít.

409 Operaci nelze provést, protože již běží jiná operace nad stejným objektem.

412 Vraceno v případě již neplatného jobId, který je povinně předáván některým endpointům.

500 Nastala neočekávaná chyba uvnitř manažera stavu.

A.3 Zdroje

Z1: GET cdimages[/<id>]

Popis:
Vrátí seznam dostupných obrazů CD. Pokud je specifikováno id vrátí pouze jeden záznam, nebo chybu 404.

Pole:
name: název obrazu CD
description: volitelné, popis obrazu CD

Příklad A.1: Odpověd serveru na GET /v1/cdimages

```
[
    {
        "name": "Debian 7 netinstall",
        "description": "Debian 7 installation cd",
        "id": "5467656d79cad154165ee72d"
    },
    {
        "name": "System Rescue CD",
        "description": "Live Linux distribution.",
        "id": "5467658c79cad154165ee72e"
    },
    {
```

```
        "name": "Debian 8 (jessie) netinstall",
        "description": "Debian 8 installation cd",
        "id": "5537b07797c2ee28ebd53bb2"
    }
]
```

Z2: GET templates[/<id>]

Popis:
Vrátí seznam dostupných obrazů templatů. Pokud je specifikováno id vrátí pouze jeden záznam, nebo chybu 404.

Pole:
fileName: Název souboru s templatem jak je uložen na salt serveru.
installScript: Salt script pro post instalační úpravy konfigurace (např. nastavení hostname). V současné době neimplementováno.
imageType: Typ templatu. Slouží saltu k určení nutných operací při nasazování.

Příklad A.2: Odpověd serveru na GET /v1/templates

```
[
    {
        "name": "Debian 8",
        "description": "GNU/Linux distribution Debian 8 (Jessie)",
        "fileName": "debian-jessie.dd.gz",
        "installScript": "debian.sls",
        "imageType": "dd.gz",
        "id": "5467651f79cad154165ee72b"
    }
]
```

Z3: GET hgroups[/<id>]

Popis:
Vrátí seznam dostupných skupin výpočetních nodů. Pokud je specifikováno id vrátí pouze jeden záznam, nebo chybu 404.

Pole:
locked: Pokud je true tak není možné do skupiny přidávat nové virtuální stroje.
metrics: Metriky výkonu. V současné době neimplementováno, vrací statická data.
storages: Seznam dostupných datových úložišť.

storages.type: Typ úložiště. Momentálně implementované jsou LVM a ZFS.

storages.params: Specifické parametry pro salt. Viz implementace úložišť v saltu.

storages.metrics: Metriky výkonu a obsazenosti. V současné době neimplementováno, vrací statická data.

Příklad A.3: Odpověd serveru na GET /v1/hgroups

```
[
    {
        "name": "Xen testing group",
        "description": "First group in S.I.C. basement",
        "locked": false,
        "metrics": {
            "ramUsed": 10,
            "ramTotal": 14,
            "cpuLoad": 10
        },
        "storages": [
            {
                "name": "local LVM",
                "description": "Lokalni LVM",
                "type": "LVM",
                "metrics": {
                    "sizeUsed": 50,
                    "sizeTotal": 100
                }
            },
            {
                "name": "local ZFS",
                "description": "Lokalni ZFS",
                "type": "ZFS",
                "metrics": {
                    "sizeUsed": 50,
                    "sizeTotal": 100
                }
            }
        ],
        "id": "5467537f79cad114163ee769"
    }
]
```

Z4: GET vms[/<id>]

Popis:

Vrátí seznam všech virtuálních strojů, nebo jeden konkrétní, pokud je specifikováno id.

Pole:

name: Název virtuálního stroje.

hypervisor: Id výpočetního nodu, na kterém se virtuální stroj nachází. Pouze pro čtení.

hgroupseq: Zaručeně unikátní číselné id VM v rámci skupiny hypervizorů. Generuje manager stavů. Pouze pro čtení.

hgroup: Id skupiny výpočetních nodů, ve které se virtuální stroj nachází. Pouze pro čtení.

state: runnning/stopped, informace o tom zda stroj běží nebo nikoli.

VNCPassword: heslo pro přístup k VNC stroje. Není určeno pro koncové uživatele, ale pro VNC koncentrátor. Zabezpečení VNC pouze heslem je nedostatečné pro veřejné sítě. Pouze pro čtení.

VNCAddress: Adresa pro přístup k VNC. Pouze pro čtení.

VNCPort: Port pro přístup k VNC. Pouze pro čtení.

ram: Množství přidělené paměti RAM v MB.

cpuno: Počet procesorů, ke kterým má VM přístup.

cdrom: Id připojeného obrazu CD. Nepovinné.

job: Informace o běžících operacích. Vše pouze pro čtení.

job.name: Název poslední provedené (nebo běžící) operace.

job.status: requested/running/finished/failed, stav poslední operace. Novou operaci nelze spustit, dokud aktuální neskončí (finished/failed).

job.diff: Popis změn, které operace vykonává ve formátu JSON PATCH (rfc6902).

job.messages: Pole stringů, může obsahovat zprávy o postupu vykonávání operace. V současné době ale není předávání zpráv implementováno.

job.id: Id operace.

nics: Pole s popisem síťových karet.

nics.mac: MAC adresa síťové karty. Generuje manažer stavu. Pouze pro čtení.

nics.vlan: 802.1Q VLAN, do které je síťová karta přiřazena.

nics.ip: IP ze které je povolen odchozí provoz.

nics.subnet: Maska podsítě.

nics.gateway: Defaultní brána (IP routeru).

hdds: Pole s popisem virtuálních disků. Na pořadí záleží, bootuje se vždy z prvního disku.

hdds.hgroupseq: Zaručeně unikátní číselné id disku v rámci skupiny výpočetních nodů. Generuje manažer stavů. Pouze pro čtení.

hdds.storage: Typ storage na kterém se disk nachází. Odkazuje se na typy storage definované ve skupině výpočetních nodů.

hdds.size: Velikost disku v GB.

Příklad A.4: Odpověd serveru na GET /v1/vms/5538bc88357e80175588b642


```
{
    "hgroupseq": 37,
    "hypervisor": "5468a05dc24342bdd8d40fa2",
    "hgroup": "5467537f79cad114163ee769",
    "state": "running",
    "name": "Test01",
    "VNCPassword": "1cc6bf5f",
    "ram": 3072,
    "cpuno": 3,
    "cdrom": "5467658c79cad154165ee72f",
    "job": {
        "name": "modify",
        "status": "finished",
        "diff": [
            {
                "op": "replace",
                "path": "/state",
                "value": "running"
            },
            {
                "op": "add",
                "path": "/hdds/0",
                "value": {
                    "size": 20,
                    "storage": "local LVM",
                    "_id": "55396f5594d573017311f4d6",
                    "placeholder": false
                }
            }
        ],
        "messages": [],
        "id": "55396f5594d573017311f4d7"
    },
    "nics": [
        {
            "mac": "fe:98:4a:55:27:d6",
            "description": "public ip",
            "vlan": 42,
            "gateway": "185.88.72.254",
            "subnet": "255.255.254.0",
            "ip": "185.88.72.4",
            "id": "5538bc88357e80175588b644"
        }
    ],
    "hdds": [
        {
            "hgroupseq": 28,
            "storage": "local LVM",
            "size": 20,
            "id": "55396f5594d573017311f4d6"
        }
    ],
```

```
    "VNCAddress": "81.0.208.41",
    "VNCPort": 5937,
    "id": "5538bc88357e80175588b642"
}
```

Z5: POST vms

Popis:

Vytvoří nový virtuální stroj. Ten bude nejdříve prázdný (žádné disky, žádné cpu, žádné síťové karty, ...). Automaticky ale bude nastartována úloha s jeho doplněním na požadovaný stav. Většina polí je identických jako u čtení VM a nebudu je zde opakovat. Pole pouze pro čtení se v požadavku nesmí vyskytovat. Navíc se objevuje pole template u disků a pole state umožňuje přijímat několik dalších hodnot. Na úspěšný POST server odpoví kódem 201 a v hlavičce Location pošle URL nového VM.

Pole:

state: runnning/stopped/restarting/destroying, umožňuje provést na VM požadovanou operaci (nastartovat, vypnout, restartovat, natvrdo vypnout)

hdds.template: Aplikuje na disk template s tímto id.

Příklad A.5: Payload dotazu POST /v1/vms

```
{
  "hgroup":"5467537f79cad114163ee769",
  "state": "running",
  "name": "testvm",
  "ram": 1024,
  "cpuno": 3,
  "cdrom": "5467658c79cad154165ee72e",
  "nics": [],
  "hdds": [{
    "size":5,
    "storage":"local ZFS",
    "template":"5467658c79dad854aaaee72a"
  }]
}
```

Z6: PUT vms/<id>

Popis:

Upraví existující VM. Struktura payloadu je identická jako u vytváření VM. Template je možné aplikovat, pouze pokud je stroj vypnutý. Požadavek selže, pokud už existuje běžící operace měnící vybraný stroj.

Z7: PUT vms/<id>/state

Popis:
Operace identická s PUT vms/<id> s tím rozdílem, že nastavuje pouze pole state.

Příklad A.6: Payload dotazu PUT /v1/vms/5538bc88357e80175588b642/state

```
{
  "state": "stopped",
}
```

Z8: POST vms/<id>/job

Popis:
Tuto operaci používá konfigurační server pro aktualizaci stavu jobu. Jakmile je stav jobu jednou nastaven na finished nebo failed, nelze ho měnit. Pro přijetí operace je nutné spolu s payloadem předat hlavičku JobId nastavenou na id jobu (viz GET vms/<id>)

Pole:
state: runnning/finished/failed. Operace běží/je úspěšně dokončená/je neúspěšně dokončená.

Příklad A.7: Payload dotazu PUT /v1/vms/state

```
{
  "state": "stopped",
}
```

Z9: DELETE vms/<id>

Popis:
Smaže VM.

Z10: GET hexport/<saltname>

Popis:
Tuto operaci používá konfigurační server pro získání dat o celém jednom výpočetním nodu. Data o VM získaná přes tento endpoint odpovídají „chtěnému“ stavu. Všechny patche u probíhajících jobů jsou aplikovány.

Pole:

saltname: Jméno, pod kterým zná výpočetní node salt. Většinou odpovídá hostname stroje.

hgroup: Do jaké skupiny výpočetní node patří.

VNCAddress: Kontaktní adresa pro VNC všech VM na tomto nodu.

VNCPortRangeStart: Od kterého jakého čísla začínají VNC porty na stroji. VNCPort VM je vypočítán z tohoto pole přičtením hgroup-seq.

vms: Kompletní seznam VM na stroji. Jednotlivé VM jsou identické jako při GET vms ale diffy u pending jobů jsou aplikovány.

storages: Seznam na nodu dostupných datových úložišť.

storages.type: Typ úložiště. Momentálně implementované jsou LVM a ZFS.

storages.params: Specifické parametry pro salt. Viz implementace úložišť v saltu.

Příklad A.8: Odpověď serveru na GET /v1/hexport/bc-test-x1

```
{
    "saltname": "bc-test-x1",
    "hgroup": "5467537f79cad114163ee769",
    "VNCAddress": "81.0.208.41",
    "VNCPortRangeStart": 5900,
    "vms": [
        ...
    ],
    "storages": [
        {
            "name": "local LVM",
            "description": "Lokalni LVM",
            "type": "LVM",
            "params": {
                "vgName": "raid1lvm"
            }
        },
        {
            "name": "local ZFS",
            "description": "Lokalni ZFS",
            "type": "ZFS",
            "params": {
                "zpool": "mainStorage",
                "zvolProperties": [
                    {
                        "name": "cz.bigcloud:autoclean",
                        "value": "on"
                    },
                    {
                        "name": "cz.bigcloud:deleted",
                        "value": "off"
```

```
                    }
                ]
            }
        }
    ],
    "id": "5468a05dc24342bdd8d40fa2"
}
```

Příloha B

Obsah přiloženého CD

- `readme.pdf` ..... tento popis obsahu CD
- `bootserver` ..... konfigurační soubory a scripty bootovacího serveru
- `json-patch-mongoose` .... fork modulu json-patch s podporou Mongoose
- `perftest` ..... data z testů výkonu v testovacím prostředí
- `promise_example` ..... příklad třídy Promise použitý v textu
- `SaltMaster` ..... implementace konfiguračního serveru
- `StateManager` ..... implementace manažera stavu
- `text` ..... text práce ve formátu LaTeX
- `DP_Gregor_Petr_2015.pdf` ..... text práce ve formátu PDF