VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SPRÁVA LOKÁLNÍCH A DISTRIBUOVANÝCH SYSTÉMŮ V CLOUDU

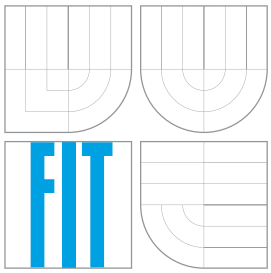
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADEK NOVÁČEK

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SPRÁVA LOKÁLNÍCH A DISTRIBUOVANÝCH SYSTÉMŮ V CLOUDU

LOCAL-BASED AND CLOUD-BASED SYSTEMS MANAGEMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. RADEK NOVÁČEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JOZEF MLÍCH
SUPERVISOR

BRNO 2011

Abstrakt

Diplomová práce se zabývá analýzou existujících řešení a návrhem architektury nového řešení pro správu unixových systémů. Vytvořený systém je použitelný jak pro konfiguraci lokálních systémů přes sběrnici D-Bus, tak i vzdáleně, kdy konfigurovaný systém je dostupný přes síť. Při návrhu systému byl brán ohled na použitelnost při cloud computingu.

Abstract

The master thesis deals with analysis of existing solutions and architecture design of new solutions for management of unix-like systems. The resulting system is usable for configuration of the local system over the D-Bus and remotely over the network. Usability for cloud computing has been considered during design.

Klíčová slova

Správa, konfigurace, D-Bus, QMF, cloud, rozhraní, Augeas.

Keywords

Management, configuration, D-Bus, QMF, cloud, interface, Augeas.

Citace

Radek Nováček: Správa lokálních a distribuovaných systémů v cloudu, diplomová práce, Brno, FIT VUT v Brně, 2011

Správa lokálních a distribuovaných systémů v cloudu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jozefa Mlícha.

.....
Radek Nováček
23. května 2011

Poděkování

Děkuji společnosti Red Hat za poskytnuté zdroje a Ing. Jaroslavu Řezníkovi za odbornou konzultaci.

Obsah

Slovník pojmů

AMQP	Advanced Message Queuing Protocol je otevřený standard pro zasílání zpráv
D-Bus	systém pro zasílání zpráv na lokálním systému
FMCI	Fedora Management and Configuration Infrastructure je projekt, jehož cílem je řízení vývoje konfiguračních nástrojů ve Fedoře
GLib	multiplatformní knihovna poskytující prostředky pro usnadnění vývoje aplikací
IPC	Inter-Process Communication – komunikace mezi procesy
QMF	Qpid Messaging Framework je sběrnice pro řízení zasílání zpráv pomocí systému QPid
Qpid	systém pro zasílání zpráv, implementuje standard AMQP
PAM	Pluggable Authentication Modules je mechanizmus pro integraci autentizačních API
RPC	Remote Procedure Call – vzdálené volání procedur
SSL	Secure Sockets Layer – zabezpečení komunikace pomocí šifrování a autentizace
XML	Extensible Markup Language je obecný značkovací jazyk
XSL	Extensible Stylesheet Language je jazyk pro popis transformací XML souborů

Kapitola 1

Úvod

S rozvojem informačních technologií je stále vyšší poptávka po správcích počítačových systémů, ať už serverů nebo jednotlivých pracovních stanic. Nakonfigurování systému je však nelehký úkol, který vyžaduje, aby administrátor měl detailní znalosti o systému i aplikacích, které na něm běží. Mezi jeho úkoly patří zajistit vysokou dostupnost poskytovaných služeb, zabezpečení proti útokům i nastavení jednotlivých programů. Po odbornících, kteří tyto úkoly zvládnou, je velká poptávka a jsou i odpovídajícím způsobem nároční finančně. Mnoho firem proto vyžaduje, aby systémy byly jednodušší, snadněji konfigurovatelné a umožnily i méně zkušeným administrátorům systém správně nastavit a zabezpečit. Z tohoto důvodu je zde neustálý tlak na vytváření nových rozhraní a konfiguračních nástrojů.

Další koncept, kterému je nutné přizpůsobit návrh nových konfiguračních rozhraní a nástrojů, je tzv. *Cloud computing*. Jedná se o přesun výpočetních zdrojů a aplikací z lokálních počítačů na vzdálené servery přes Internet. Na tyto servery jsou kladeny odlišné nároky od běžných serverů, například je nutná maximální flexibilita a škálovatelnost, což opět vyžaduje jiný přístup ke konfiguraci.

Jedním z důvodů, proč tato práce vznikla, je stav konfiguračních nástrojů v linuxové distribuci Fedora. Mimo obecných nástrojů, které jsou součástí většiny distribucí GNU/Linuxu, ve Fedoře existují tzv. system-config nástroje, které byly vytvořeny specificky pro tuto distribuci. Některé z těchto nástrojů jsou bohužel ve špatném stavu, kdy nejsou udržované a nereflektují změny v programu, který konfigurují. Tyto nástroje jsou vytvářeny samostatně různými vývojáři bez jakékoli koordinace, proto různě vypadají a různě se ovládají. Z tohoto důvodu vznikla iniciativa *system-config cleanup* [57], jíž jsem součástí. Jejím cílem je sjednotit rozhraní těchto programů, opravit chyby a zbavit se nutnosti běhu se zvýšenými oprávněními. Dále se ukázalo, že některé nástroje buď úplně chybí nebo je nutné je přepsat.

Cílem této práce je vytvořit nástroj, který usnadní lokální i vzdálenou správu unixových systémů. Tento nástroj bude poskytovat rozhraní, které bude snadno použitelné dalšími nástroji pro konfiguraci a monitorování systémů.

Klíčovou součástí nástrojů pro správu systémů je komunikační rozhraní. V kapitole 2 jsou diskutovány technologie, které budou v implementační části práce použity pro komunikaci mezi jednotlivými částmi systému. Jsou zde popsány systémy pro komunikaci mezi aplikacemi jak na lokální úrovní v rámci jednoho počítače, tak i vzdáleně přes síť. Také jsou zde uvedeny existující nástroje pro správu systémů a komunikační rozhraní, které využívají.

Další kapitola je věnována *cloud computingu* a příkladům nejznámějších cloudů. Jedním z požadavků navrženého systému je i zjednodušení konfigurace pro tyto systémy, které zažívají v současné době velký rozmach.

Navržený systém je rozdělen do několika částí. První část bude platformě nezávislá

knihovna, která bude vykonávat vlastní správu systému. Nad touto knihovnou běží systémoví démoni (tzv. agenti), kteří komunikační rozhraní zpřístupní jak pro lokální, tak i pro vzdálené nástroje. Toto rozhraní pak bude využito programy s grafickým uživatelským rozhraním. Detailní popis architektury je v kapitole 4.

V kapitole 5 jsou popsány detaily implementace daného systému. Jsou zde diskutovány problémy, které nastaly při vytváření knihovny tvořící rozhraní se systémem. Pro zjednodušení vytváření dalších agentů je značná část kódu pro vytvoření jejich rozhraní automatizována. Další důležitá část, která je v této kapitole popsána, je zabezpečení systému proti neoprávněnému užití.

Každý počítačový systém musí být před nasazením důkladně otestován. Mimo testů vývojáři v průběhu vývoje je vhodné, aby byl výsledný systém otestován jako celek. Je však velice obtížné zajistit, aby byly otestovány všechny části systému, a to nejlépe po každé změně. Značným zlepšením je automatizace této úlohy. Spuštění sady testů po každé změně v programu může přinést značné zlepšení v oblasti zajištění kvality aplikace. Nicméně je nutné, aby tato sada testů byla dobře napsána a její výsledky vhodně interpretovány. Testováním systému vytvořeného v rámci této práce se zabývá kapitola 6.

Poslední kapitola nastiňuje možné další pokračování vytvořeného programu v budoucnu. Vzhledem k tomu, že je jedná spíše o platformu pro vytváření dalších modulů, není vývoj zdaleka u konce. Je zde popsáno několik možností dalšího rozvoje, z nichž některé budou v nejbližší době vybrány a vypracovány.

Tato diplomová práce navazuje na semestrální projekt, ze kterého přebírá informace a dále je rozšiřuje. Části kapitol 2 až 5 jsou převzaty ze semestrálního projektu.

Kapitola 2

Systémová rozhraní pro správu

Tato kapitola se zabývá vymezením terminologie související s komunikací jednotlivých systému a existujícími nástroji pro správu. Podrobněji jsou diskutovány komunikační systémy relevantní k diplomové práci. V první části je rozebrán systém komunikace D-Bus, přičemž základní informace byly čerpány z [25, 51, 26] a popis objektového modelu z [3, 22]. Dále je zde diskutován systém QMF, jehož popis a princip funkce byl čerpán z [47]. Obecné informace o komunikaci mezi procesy byly převzaty z [50]. V některých případech běží systémy, které mezi sebou komunikují, na různých úrovních oprávnění. Je tedy vhodné mít prostředek, který umožní ověřit, zda systém na nižší úrovni má oprávnění požadovat vykonání akce na systému s vyšším oprávněním. Systém PolicyKit může využít svých vnitřních pravidel a interakce s uživateli k ověření, zda je takováto eskalace oprávnění přípustná. Principy PolicyKitu jsou převzaty z [63]. Pro srovnání jsou v této kapitole také uvedeny existující nástroje pro správu a způsob jejich komunikace.

Pro potřeby tohoto projektu je nutné mít dvě nezávislá komunikační rozhraní — pro lokální správu a pro správu vzdálenou. Obě varianty mají svá specifika, se kterými je nutno počítat a přizpůsobit jim návrh rozhraní. V obou případech je nutné zaručit několik požadavků, které musí systémy splňovat. Je nutné, aby byly všechny zaslané zprávy doručeny a nedošlo k jejich ztrátě. To je zaručeno posíláním potvrzení o přijetí zprávy. Dále je třeba zaručit, že nedošlo k modifikaci zprávy při přenosu. K tomu se využívají například kontrolní součty, které se mohou ještě kombinovat se samoopravnými kódy, které dokáží částečně poškozenou zprávu opravit. Při komunikaci přes síť se o tento požadavek zpravidla starají nižší vrstvy síťového protokolu.

Pokud má být daný systém pro komunikaci mezi procesy dostatečně výkonný a robustní, je třeba vyřešit zpracování více požadavků souběžně. Běžně systémy využívají tří přístupů k tomuto problému. První z nich využívá vlákna. Při příchodu každého požadavku je třeba vytvořit nové vlákno, které tento požadavek zpracuje a poté se ukončí. Nevýhoda tohoto systému však je nutnost koordinace přístupu ke sdíleným objektům, což může vést k synchronizačním problémům. Některé implementace vláken také přináší zvýšení režie, které se negativně projeví na výkonnosti celého systému.

Další přístup ke zpracování souběžných požadavků je řízení událostmi. Každý požadavek je uložen jako událost do fronty, která je postupně zpracována. Pokud není v této frontě žádná událost, systém pasivně čeká. Výhoda tohoto přístupu je v jednodušší implementaci, protože není nutné řešit problémy paralelního přístupu a na jednoprocesorových systémech dosahuje tento přístup vyššího výkonu, protože není nutné přepínání kontextu mezi jádrem a uživatelským prostorem. Nevýhoda je nemožnost využít více jader nebo procesorů, což může systém při velkém zatížení urychlit.

Poslední zde uvedený přístup je využití procesů, kdy je každý požadavek zpracován jedním procesem a poté je proces ukončen. Kvůli vysoké režii vytvoření procesu (je nutné zkopírovat paměťový prostor programu) je vhodné proces neukončovat, ale umístit jej do fronty nezaneprázdněných procesů, ze které se pouze vybere proces při příchodu požadavku. Tento přístup také vyžaduje synchronizaci a mechanizmus pro sdílení dat, ale v některých případech to není vyžadováno, což činí tento přístup vhodný pro jednoduché účely jako například webový server pro statické stránky.

Jedna z věcí, které je nutno při návrhu aplikace rozhodnout, je použitý systém pro komunikaci mezi procesy. Protože lokální a vzdálená zpráva mají často velmi odlišné požadavky, je možné použít pro obě oblasti různé systémy.

De-facto standardem pro použití na linuxovém desktopu je systém D-Bus, jenž je součástí běžné instalace většiny distribucí. Také jej využívá mnoho aplikací, které poté mohou využít implementované rozhraní pro svoje potřeby.

Pro správu vzdálenou není žádný komunikační systém, jehož použití by výrazně převažovalo. Existuje několik běžně používaných systému, mezi ty nejznámější patří XML-RPC, CORBA, JSON-RPC, AMQP a další. Pro potřeby tohoto projektu byl vybrán systém AMQP, konkrétně implementace Qpid a jeho nadstavba QMF.

2.1 D-Bus

D-Bus (Desktop Bus) je systém pro předávání zpráv mezi procesy (IPC[1]) a je vyvíjen pod záštitou projektu *freedesktop.org*. Tento systém byl ovlivněn systémem DCOP z desktopového prostředí KDE (verze 2 a 3), který v KDE verze 4 nahradil. D-Bus kromě zasílaní zpráv také obsahuje mechanizmy pro spouštění aplikací a démonů na vyžádání. Původně byl napsán pro operační systém GNU/Linux, ale je portován i na ostatní UNIX-like systémy. V současné době se pracuje i na portu na Microsoft Windows.

Typicky D-Bus obsahuje dva typy sběrnic (Bus). Na perzistentní systémovou sběrnici (SystemBus), která je spuštěna při startu systému a používá ji operační systém spolu s démony pro vzájemnou komunikaci, jsou kladena přísnější bezpečnostní omezení. Z toho důvodu jsou potřeba relační (SessionBus) instance, které běží pro každého přihlášeného uživatele a umožňují komunikaci bez omezení [25]. Všechny sběrnice je také možno adresovat pomocí cesty k jejímu soketu na disku, např. `unix:path=/var/run/dbus/system_bus_socket`. Převzato z oficiálních stránek projektu D-Bus [51].

Základní schématem pro popis rozhraní D-Busu je objektový model, který slouží pro popis a identifikaci jednotlivých rozhraní a jejich hierarchii. Každá zpráva zaslaná přes D-Bus má zdroj a cíl, které se v této terminologii nazývají *objekty*. Každá aplikace může obsahovat několik objektů — zdrojem a cílem zpráv jsou právě tyto objekty nikoli aplikace, které je poskytují. Objekt je identifikovaný jeho *cestou*, která má tvar jako běžná cesta v unixových systémech — začíná lomítkem a jednotlivé úrovně jsou také odděleny lomítkem. Příkladem takové cesty může být:

```
/org/fedoraproject/fmci/SomeObject
```

Dalším důležitým pojmem je *název připojení* (angl. Connection name), které určuje, kam se má zpráva doručit. Jsou dva typy názvu připojení, a to unikátní ve tvaru například `:1.123` nebo tzv. „well known“, například:

```
org.fedoraproject.fmci
```

[1] z anglického IPC – InterProcess Communication – meziprocesní komunikace

Navíc každý objekt může mít několik *rozhraní* (angl. Interface), což je popis, které zprávy je možno objektu zaslat. Rozhraní má stejný formát jako název připojení (často jej obsahuje jako svůj prefix). Příkladem rozhraní může být například:

```
org.fedoraproject.fmci.SomeInterface
```

Většina objektů implementuje několik standardních rozhraní, které zjednodušují práci s objektem. Rozhraní `org.freedesktop.DBus.Introspectable` slouží pro zjištění, co objekt poskytuje. Součástí tohoto rozhraní je jediná metoda `Introspect`, která je bez parametrů a vrací XML popis všech rozhraní, které daný objekt poskytuje. Toho mohou využít různé podpůrné nástroje pro vizualizaci objektů, například D-Feet nebo qdbusviewer. Dalším standardním rozhraním je `org.freedesktop.DBus.Properties` pro přístup k vlastnostem objektu, které má tři metody `Get`, `Set` a `GetAll`. První z nich slouží ke zjištění vlastnosti objektu, druhá k nastavení. Metoda `GetAll` vrací slovník, kde jako klíče jsou názvy všech vlastností a k nim jsou přiřazeny jejich hodnoty.

Rozlišují se čtyři druhy *zpráv* — volání metody, návrat z metody, signál a chyba. Zpráva o volání metody je předána cílovému objektu, který ji zpracuje a odpoví zprávou návrat z metody nebo chybovou zprávou. Signál probíhá v opačném směru. Objekt vyšle signál, který je poté předán všem objektům, které se zaregistrovaly k jeho odebírání. Každá zpráva může volitelně obsahovat libovolný počet argumentů, které jsou silně typované a obsahují jak primitivní typy (boolean, integer, float), tak i strukturované (string, array, dictionary).

Pro vyzkoušení volání metody je možno použít utilitu *dbus-send*. K vypsání seznamu všech připojení na systémové sběrnici je možno použít následující příkaz:

```
dbus-send --system --print-reply --dest=org.freedesktop.DBus \
/ org.freedesktop.DBus.ListNames
```

Vzhledem k tomu, že se u systému D-Bus nepožaduje vysoký výkon, ale spíše jednoduchost a nenáročnost na systémové prostředky, používá řízení událostmi pro zpracování požadavků. Každé spojení D-Bus serveru s klientskou aplikací má dvě fronty (vstupní a výstupní), ve kterých jsou uloženy nezpracované události. Tyto události jsou zpracovány v hlavní smyčce. Pokud není třeba zpracovat žádnou událost, hlavní smyčka pasivně čeká na objevení nové události.

D-Bus používá binární protokol pro přenos zpráv, zakódování a dekódování dat je tedy rychlejší než u protokolů založených na textu. Zpráva je rozdělena na hlavičku obsahující metadata a data samotné. Metadata obsahují informace o odesilateli a příjemci zprávy a signaturu typů, která je použita při dekódování zprávy.

Mezi užitečné vlastnosti systému D-Bus patří aktivace služeb na požádání. V konfiguračním souboru je uvedeno jméno připojení a cesta k programu, který toto připojení poskytuje. Pokud nějaký proces požádá sběrnici D-Bus o vytvoření spojení s neexistujícím název připojení, je na základě těchto konfiguračních souborů zjištěno, zda se má spustit asociovaný program. Není tedy nutné, aby služby běžely neustále, což snižuje požadavky na systémové prostředky.

Více podrobností o systému D-Bus lze nalézt v [3], [22] a [35], odkud bylo čerpáno.

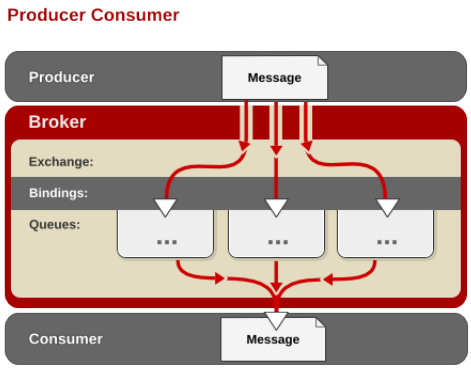
2.2 QMF

V této podkapitole bude popsán framework QMF včetně technologií, na kterých je postaven [46].

AMQP, Advanced Message Queuing Protocol, je otevřený protokol pro zasílání zpráv, jehož specifikace vytváří skupina AMQP Working Group. Cílem tohoto projektu je vytvořit infrastrukturu pro zasílání zpráv mezi aplikacemi, která bude naprosto otevřená, platformě nezávislá, použitelná pro široké spektrum aplikací a umožní spolupráci mezi různorodými aplikacemi [47].

Protokol AMQP je rozdělen do několika oddělených vrstev. Na nejnižší vrstvě je definován efektivní binární peer-to-peer[2] protokol pro přenos zpráv mezi dvěma procesy po síti. Další vrstva je abstraktní formát zpráv s konkrétním standardním kódováním. Každý proces vyhovující AMQP musí být schopen přijmout a odeslat zprávy v tomto standardním kódování [47, str. 7].

AMQP server, často nazývaný *Broker*, se skládá ze tři komponent, viz obrázek 2.1. „Exchange“ přijímá zprávy a předává je do front zpráv, jestliže splňují kritéria definovaná v části „bindings“. Ta definuje vztah mezi „exchange“ a frontou (queue), do které má být zpráva směrována. Ve frontě zpráv jsou uloženy zprávy a odtud jsou doručeny příjemcům, kteří si zaregistrují příjem z dané fronty [2].



Obrázek 2.1: Schéma AMQP brokeru, převzato z [2]

Existuje několik schémat pro zasílání zpráv: přímé (doručení zprávy do určité fronty), publish-subscribe (zaslání zprávy do všech front, které se zapsaly k odběru) a XML (všechny zprávy, které odpovídají dotazu ve formátu XPath jdou do dané fronty) [21].

Apache Qpid je implementace specifikace AMQP vyvíjená pod záštitou The Apache Software Foundation. Součástí jsou dvě implementace Brokeru – v C++ a v Javě a bindingy[3] pro jazyky Java, C++, C# .NET, WCF Adapter, Python a Ruby. Cílem projektu je stoprocentní kompatibilita s protokolem AMQP. Qpid je vyvíjen pod licencí Apache License, verze 2.0h.

Veškerá komunikace mezi klientem a brokerem může být šifrována pomocí SSL/TLS. C++ broker také podporuje šifrování pomocí systému Kerberos, což však nepodporují někteří klienti. Pro autentizaci je možné využít SASL, což je systém pro ověření klientů na serverech. Po navázání šifrovaného spojení a autentizaci klientské aplikace je třeba ověřit, zda má klient právo na provedení určité akce. Tato autorizace se řídí pomocí přístupového listu (ACL), kde se specifikován vztah mezi akcemi a rolemi jednotlivých uživatelů. Je také možné vytvářet skupiny sdílející nastavení oprávnění.

Vzhledem k tomu, že se u systému Qpid klade velký důraz na výkonnost, je nutné

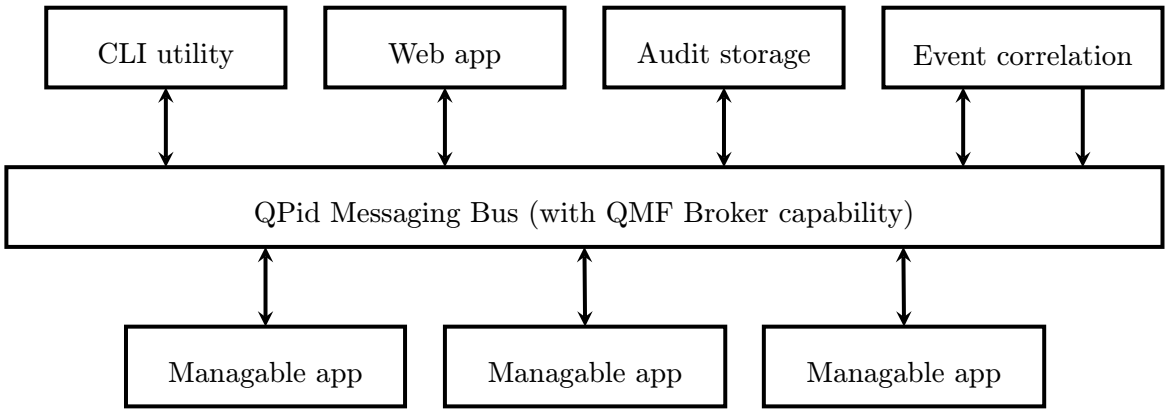
[2]Peer-to-peer je přenos dat mezi dvěma uživateli přímo bez serveru

[3]Language binding je specifikace standardního rozhraní k určitému prostředku v daném jazyce [17]

paralelní zpracování požadavků. Souběžnost je založena na vláknech. Každý požadavek na zpracování je indikován příchozím signálem. Jedno vlákno tento požadavek zpracuje a poté je vráceno do seznamu neaktivních vláken, kde čeká na probuzení další úlohou. Tímto je umožněno efektivní využití vícejádrových a víceprocesorových systémů. Ve výchozím stavu je počet vláken o jedno vyšší, než je celkový počet procesorových jader. Fakt, že jsou vlákna přidělena na zpracování jednotlivých požadavků a nikoli pro interakci s klientem přímo, umožňuje obsluhovat libovolný počet klientů současně.

Qpid je využit i v produktu MRG firmy Red Hat. Jedná se kombinaci zasílání zpráv pomocí Qpid s upraveným jádrem pro podporu reálného času a nástrojů pro vysoce náročné výpočty. Použitím realtime jádra je dosaženo zajištění odezvy v konstantním čase nezávisle na počtu přijatých zpráv za určitý čas. Snížením latencí funkcí pro obsluhu přerušení a zlepšením řízení priorit jednotlivých částí systému je dosaženo snížení nedeterminizmu, což vede k rychlejší době obsluhy požadavků. Je možné například upřednostnit síťová rozhraní nad vstupně–výstupním operacemi [7].

QMF je zkratka z anglického Qpid Management Framework. Jedná se víceúčelovou sběrnici postavenou na systému pro zasílaní zpráv Qpid. Mezi hlavní výhody patří škálovatelnost, bezpečnost a funkčnost, kterou poskytuje použití Qpidu jako základu. Na obrázku 2.2 je schéma systému postaveného na frameworku QMF. Systém QMF má tři základní komponenty: *konzole*, *agent* a „*broker*“. Konzole slouží řízení agentů. může to být například CLI aplikace, webová aplikace nebo systém pro monitorování agentů. Agent je jakýkoli program, který je možné řídit pomocí QMF. Broker je centrální řídicí bod komunikace, většinou je shodný s Qpid brokerem. V tomto schématu jsou tři agenti označení *Manageble apps*. *CLI utility* značí aplikaci s textovým uživatelským rozhraním, *Web app* je webová aplikace a *Audit storage* je služba pro ukládání zpráv o běhu, z pohledu názvosloví QMF jsou všechny tři aplikace konzole. *Event correlation* je aplikace, která agreguje data z agentů. Je to tedy jak konsole i agent současně, protože vytváří nová data (funkce agenta) a zároveň se dotazuje ostatních agentů (funkce konsole).



Obrázek 2.2: Schéma systému pracujícího nad QMF frameworkem [46].

Agent v kontextu QMF neodpovídá inteligentnímu agentu. Dle Jenningse [19] má být inteligentní agent autonomní, reaktivní, proaktivní a má mít sociální schopnosti. QMF agent nemusí tyto podmínky splňovat. Například sociální schopnosti agent většinou nemá, protože není schopen spolupráce s ostatními agenty a omezuje se jen na vzájemné voláni funkcí. Je však možné pomocí QMF agentů inteligentní agenty vytvořit. Vzhledem k tomu,

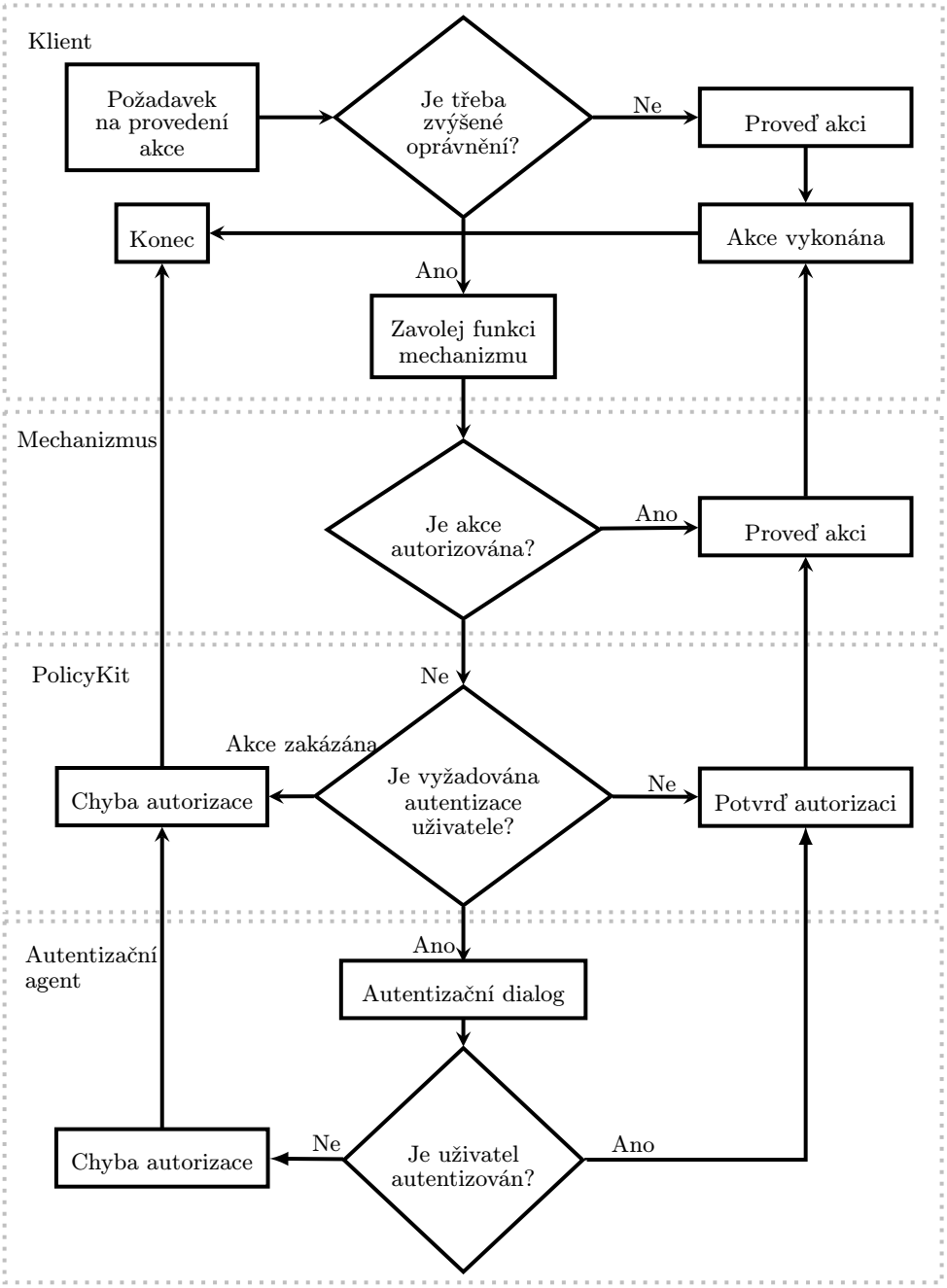
že označení agent je součástí oficiální terminologie QMF, bude tento pojem využit i v dalším textu.

2.3 PolicyKit

PolicyKit poskytuje rozhraní, které programům bez zvýšeného oprávnění umožňuje používat služby nabízené programy se zvýšenými oprávněními [38]. Schéma funkce PolicyKitu je na obrázku 2.3. Oproti běžným aplikacím, které vyžadují administrátorská oprávnění, je nutné při použití systému PolicyKit rozdělit program na dvě části. První část, označována jako mechanizmus, by měla obsahovat pouze funkce, které nutně vyžadují zvýšená oprávnění. Ve druhé části bude zbytek programu, který bude běžet pod běžným uživatelem. Na schéma 2.3 je tato část označena jako klient. Pokud klient potřebuje zavolat funkci, která implementována v mechanizmu, zavolá ji běžným způsobem. Mechanizmus se poté zeptá PolicyKit démona, zda má volající oprávnění provést danou akci. Pak proběhne ověření. PolicyKit se podívá do svých konfiguračních souborů, jaké oprávnění daná akce vyžaduje. Pokud požaduje vyšší oprávnění, než jaké má k dispozici uživatel, který se akci pokouší provést, dotáže se autentizační agent uživatel na heslo uživatele se zvýšenými oprávněními. Když je zadané heslo správné, akce se vykoná.

V konfiguračních souborech PolicyKitu je možné nastavit několik úrovní implicitních nastavení autorizací. Kromě bezpodmínečného zakázání či povolení akce je možné nastavit nutnost autorizace pro současného uživatele a pro administrátora. K těmto dvěma možnostem existují alternativy, kdy je obdržená autorizace zachována pro krátký časový úsek. To je vhodné, když je třeba nějakou akci vykonat vícekrát po sobě, kdy uživatel nemusí zadávat heslo několikrát.

V některých případech je vhodné provést několik akcí najednou, avšak zeptat se uživatele na heslo jen jednou. To není v současné době v systému PolicyKit možné. Jeden z možných případů užití je následující: Uživatel změní v grafickém nástroji nastavení programu a požaduje, aby se změna zapsala do konfiguračního souboru programu a okamžitě aplikovala na již běžící instanci tohoto programu. Pokud se tato akce rozdělí na dvě samostatné akce – zápis do konfiguračního souboru a okamžitá aplikace (například restartováním systémové služby), PolicyKit bude dvakrát zjišťovat autorizaci akce, takže se uživateli zobrazí dialog pro zadání hesla dvakrát, což je obtěžující. Jeden ze způsobů, jak toto omezení obejít, je definování nové akce, která provede oba kroky najednou. Toto řešení není příliš přívětivé pro vývojáře, protože musí implementovat program, který tyto akce na požádání vykoná (tzv. *helper*), ačkoli jejich implementace již může být dostupná přes rozhraní D-Bus. Řešením tohoto problému by bylo zavedení skupin akcí do PolicyKitu. Skupina by měla zadáno, ze kterých akcí se bude skládat a jaké oprávnění pro ni bude potřeba, případně by to mohlo být nejvyšší ze všech oprávnění akcí ve skupině. Program, který by chtěl akce z této skupiny provést, by nejprve musel zavolat funkci pro ověření autorizace pro tuto skupinu. PolicyKit by si na určitou dobu zapamatoval, že jsou tyto akce již autorizované a již by nic neověřovat, až by byl z implementace akcí zavolán. Pro zvýšení bezpečnosti by mohl program po skončení těchto akcí autorizaci zrušit. Konceptuálně by toto řešení bylo funkční, nicméně by vyžadovalo velký zásah do kódu PolicyKitu a změnu API.



Obrázek 2.3: Schéma ověření autorizace pomocí systému PolicyKit.

2.4 Existující nástroje pro správu

Tato podkapitola se zabývá přehledem vybraných nástrojů pro správu unixových systému, které jsou relevantní pro tuto práci. Tyto nástroje umožňují správcům nastavit systém a některé programy bez nutnosti ruční editace textových souborů nebo použití příkazové řádky. Součástí instalace některých služeb jsou vlastní nástroje pro konfiguraci. Například program Samba, což je svobodná implementace protokolu SMB pro vzdálený přístup k sou-

borům, obsahuje webové rozhraní zvané SWAT. Pomocí tohoto rozhraní je možné snadno nastavit sdílení souborů a tiskáren [41]. Podobně i služba CUPS, která se stará o tisk, má vestavěné webové rozhraní. Mnoho dalších programů však takové nástroje neobsahuje, proto se vytváří nástroje, které jejich konfiguraci zjednodušují a zpřístupňují i méně zkušeným administrátorům.

Základní přehled nástrojů pro správu unixových systémů je převzat z [33].

System-config je sada samostatných nástrojů vyvíjených pro distribuci Fedora. Tyto nástroje jsou mnohdy zastaralé a jedním z cílů iniciativy FMCI, zmíněné v dalším textu, je vytvořit náhradu některých z nich. Jedná se o nástroje s grafickým uživatelským rozhraním, většinou vytvořených v jazyce Python s použitím knihovny GTK pro tvorbu GUI. Vzhledem k tomu, že byly vytvořeny různými autory nezávisle, je i jejich rozhraní a způsob použití odlišný. Pracují většinou přímo s konfiguračními soubory. Při spuštění daný soubor přečtou a zpracují, při ukončení uloží změny na disk. Častým problémem bývá, že nástroje při uložení vytvoří celý konfigurační soubor, což může vést ke ztrátě určitých informací, například komentářů.

Vážný problém je to, že většinu těchto nástrojů je nutné spouštět s administrátorskými oprávněními, což přináší bezpečností riziko. Je využito nástroje *ConsoleHelper*, který se před spuštěním programu ve výchozím režimu zeptá na administrátorské heslo a poté spustí aplikaci s vyššími oprávněními. Pro autentizaci uživatelů je však možné využít i systém PAM, což je zapouzdření nízkoúrovňových autentizačních rozhraní do vysokoúrovňového API. Tím je umožněno využít i alternativní metody autentizace uživatelů jako čipové karty, otisky prstů nebo protokol Kerberos. S rostoucím množstvím kódu roste i riziko zneužitelných chyb. Tím, že program běží se zvýšenými oprávněními, jsou potenciální škody mnohem vyšší.

Jedním z možných řešení je použití nástroje *PolicyKit*, který je popsán výše. Nicméně jeho použití často vynucuje masivní zásahy do návrhu i implementace programu. Z tohoto důvodu většina autorů a současných správců těchto nástrojů nechce nebo z časových důvodů nemůže PolicyKit začlenit.

Puppet je nástroj pro administrátory, který jim umožňuje jednodušší správu systémů. Používá vlastní deklarativní jazyk na popis konfigurace systému, což umožňuje přenést nastavení na libovolný počet dalších systémů. Puppet je převážně zaměřen na unixové systémy, ale má i základní podporu pro Microsoft Windows. Je napsán v jazyce Ruby pod licencí GNU GPL [62]. Pro komunikaci mezi serverem a jednotlivými klienty používá rozhraní založené na architektuře REST, což je jednotné rozhraní pro přístup ke zdrojům v distribuovaném prostředí [39]. Protokol založený na architektuře REST je podobný HTTP, má stejný formát a základní příkazy GET, POST, DELETE a PUT, které odpovídají získání, vytvoření, smazání a změně určitých dat. Pro zabezpečení komunikace je použito SSL, které vytváří bezpečný tunel mezi serverem a klientem.

Spacewalk je webová aplikace umožňující spravovat linuxové operační systémy. Jedná se open-source verzi nástroje Red Hat Network Satellite. Spacewalk primárně slouží pro monitorování běhu systému, instalaci a aktualizaci softwarových balíků. Dále je možné tento nástroj použít pro monitorování a správu virtuálních systému [56]. Pro komunikaci je použito rozhraní postavené na XML-RPC. Jedná se o vzdálené volání procedur pomocí předávání XML souborů přes HTTP protokol. O zabezpečení komunikace se opět stará SSL.

Webmin slouží pro grafickou konfiguraci unixových systému pomocí webového rozhraní. Tento systém byl navržen tak, aby byl jednoduchý, snadno použitelný a rozšiřitelný. V současné době podporuje více než 35 různých unixových systémů [8]. Mezi možnosti nástroje Webmin patří například přidávání, odebíraní a editace uživatelských účtů, nastavení disko-

vých kvót pro jednotlivé uživatele, správa softwarových balíčků, nastavení síťových rozhraní a správa služeb, například Apache, MySQL, PostgreSQL, Samba a NFS [4]. Webmin běží lokálně na stroji, který konfiguruje a uživatelé ze připojují přes webový prohlížeč k rozhraní Webminu. Je zde opět možno zabezpečit komunikaci pomocí SSL.

Landscape vyvíjí firma Canonical a slouží pro monitorování a správu systémů s distribucí Ubuntu. Jedná se o webové rozhraní, které umožňuje monitorovat stav systému, administraci uživatelů a instalaci nových verzí softwaru. Jedná se však o proprietární aplikaci, která je k dispozici pouze uživatelům Ubuntu, kteří platí podporu od firmy Canonical [18]. Landscape používá vlastní protokol založený na HTTP s podporou SSL.

Cfengine je nástroj pro konfiguraci unixových systémů [34]. Jedná se o centralizovaný systém, kdy je konfigurace ze serveru rozesílána na jednotlivé klienty. Rozdíl v nastavení nástroje cfengine oproti ostatním spočívá v tom, že se definuje, jak se má systém chovat. U ostatních systému je definováno, co se má provést. Například pokud chce mít administrátor v souboru `/etc/hosts` nějaký specifický záznam, u běžných nástrojů provede příkaz pro přidání toho záznamu do souboru. U cfengine nadefinuje ve vestavěném jazyce, že daný systém má obsahovat daný záznam. Cfengine zjistí, zda tomu tak je, a pokud ne, záznam přidá. Výhoda tohoto přístupu je v tom, že se nejedná o jednorázovou akci zapsání informace do souboru, ale dlouhodobě se kontroluje výsledek této akce. Cfengine používá pro komunikaci mezi serverem a klienty vlastní protokol, o zabezpečení komunikace se stará opět SSL.

	Informace o HW	Správa softwaru	Konfigurace	Monitorování	Správa uživatelů	Služby	Open source
Puppet	Ne	Ano	Ano	Ano	Ano	Ano	Ano
Spacewalk	Ano	Ano	Ano	Ano	Ne	Ne	Ano
Webmin	Ne	Ano	Ano	Ano	Ano	Ano	Ano
Landscape	Ano	Ano	Ne	Ano	Ano	Ne	Ne
Cfengine	Ne	Ne	Ano	Ano	Ne	Ne	Ano
Matahari	Ano	Ne	Ne	Ano	Ne	Ano	Ano

Tabulka 2.1: Porovnání vybraných nástrojů pro správu.

V tabulce 2.1 je uveden přehled funkcí a vlastností výše uvedených nástrojů. Informace se vztahují na výchozí instalaci programu a neberou v potaz možná rozšíření, která jsou distribuovaná zvlášť. První sloupec *informace o HW* zohledňuje, zda je možné pomocí daného nástroje zjistit, na jakém hardwaru systém běží. *Správa softwaru* se zabývá možností instalovat balíčky softwaru a tím instalovat i aktualizace a opravy. *Konfigurací* je v tomto přehledu myšleno, zda je nějakým způsobem možné distribuovat na nastavované systémy konfigurační soubory. *Monitorování* se zabývá možností sledovat informace o stavu systémů, které systémy běží, jejich zatížení a stav dalších zdrojů. *Správa uživatelů* znamená, zda má daný nástroj možnost přidávat a odebírat uživatele, případně měnit jejich nastavení. *Služby* ze zabývají možností zapínání a vypínání systémových služeb a démonů. Poslední sloupec *open source* značí, zda je daný nástroj dostupný pod otevřenou licencí. U nástroje cfengine je v tomto přehledu uvedena pouze explicitní podpora pro danou funkci. Pomocí vestavěného jazyka je možné definovat libovolnou další funkci.

Pro srovnání je v tomto přehledu uveden i systém Matahari, jehož částmi návrhu a implementace se zabývá tento projekt. Stav uvedený v tomto přehledu odpovídá stavu v době odevzdání této práce. Některé další funkce tohoto systému však budou přidány v nejbližší době. Například ve sloupci *konfigurace* je uvedeno, že není součástí, avšak prototyp byl v rámci této práce vytvořen a v současné době čeká na začlenění do projektu.

Kapitola 3

Cloud computing

Tato kapitola se zabývá pojmem „Cloud computing“, jeho definicí, rozdělením a příklady. Informace jsou čerpány z [43, 12].

Cloud computing je možné chápat jako další krok ve vývoji počítačů, kdy dynamicky škálovatelné a často virtualizované zdroje jsou poskytovány jako služba přes Internet. To umožňuje uživatelům různých zařízení jako PC, notebooků, PDA a chytrých mobilních telefonů (tzv. *smartphone*), přistupovat k programům, datovým úložištím a aplikačním platformám přes Internet. Výhody cloud computingu jsou snížení nákladů, vysoká dostupnost a snadná rozšiřitelnost [12].

Rozdíl mezi cloud computingem a grid computingem nemusí být na první pohled zřejmý. V obou případech se spojení výpočetních zdrojů do skupiny za účelem dosažení společného cíle. Hlavní rozdíl spočívá jejich v zaměření.

Grid computing je rozprostření jednoho komplexního úkolu mezi počítače spojené do sítě, aby se tento problém vyřešil v co nejkratším čase [10].

Na druhé straně cloud computing je metoda poskytování zdrojů mnoha uživatelům přes Internet. Zdroje mohou zahrnovat jak software, tak operační systémy a hostingové služby. Cloud computing je tedy pružnější, protože je možné na něm provádět i úkoly pro grid computing. Častěji se však jedná o provozování jednodušších služeb pro velké skupiny uživatelů. Nicméně velké cloudy pro mnoho uživatelů bývají postaveny na výpočetních gridech, které si mezi sebou rozdělují požadavky jednotlivých uživatelů. Cloud computing je tedy s grid computingem často svázán a rozdíl je pouze v úhlu pohledu. Z pohledu provozovatele se jedná o grid computing, protože má velké množství počítačů, pomocí kterých řeší určitý problém (například zpracování požadavků od uživatelů). Pro uživatele jde o cloud computing, čili o službu dostupnou na požádání, schopnou dynamicky se přizpůsobit aktuálním požadavkům.

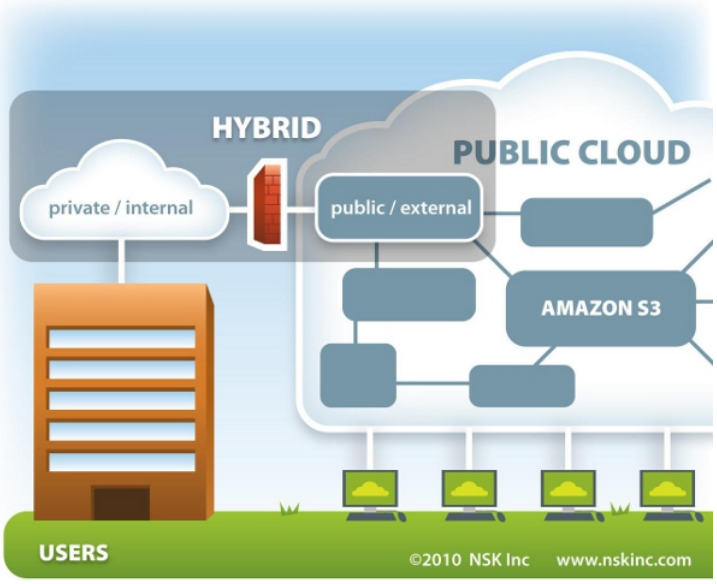
Nejznámějším případem grid computingu je program *SETI@home* [20]. Název SETI je zkratka z Search for Extra-Terrestrial Intelligence (hledání mimozemské inteligence). Cílem tohoto programu je hledání inteligentního života mimo Zemi. Jeden ze způsobů hledání je zachytávání a analýza rádiových signálů s úzkou šířkou pásma přijatých z vesmíru. Analýza těchto dat je velice náročná na výpočetní výkon, proto v roce 1999 začala Kalifornská univerzita v Berkeley využívat počítače dobrovolníků přípojené k Internetu. Tímto krokem získala výrazně vyšší výkon.

Tato kapitola se zabývá rozdělením cloud computingu ze dvou různých hledisek [5, 12, 16]. Poté jsou nastíněny technologie, nad kterými bývají cloudy postaveny [15]. Dále je zde uvedeno několik nejznámějších služeb založených na cloud computingu [23].

3.1 Dělení cloud computingu

Cloud computing můžeme rozdělit podle dvou různých hledisek. První způsob dělení je podle způsobu, jak je poskytován, druhý způsob je rozdělení podle druhu služeb, které poskytuje. Prvním způsobem dělení je možno rozdělit cloud computing na čtyři skupiny podle modelu nasazení: veřejný, soukromý, hybridní a komunitní cloud. Druhý způsob dělení je podle toho, co je nabízeno, zda hardware, software, nebo kombinace obou.

Poskytovatel *veřejného cloudu* (angl. Public cloud) nabízí své služby široké veřejnosti přes Internet. Základem je vysoká škálovatelnost nabízených služeb, mezi než patří výpočetní výkon, úložný prostor nebo softwarové služby. Tyto služby bývají často nabízeny na základě skutečně použitých zdrojů, kdy zákazník platí pouze za to, co využívá místo tradičních způsobu placení paušálních poplatků.



Obrázek 3.1: Znázornění hybridního cloudu, převzato z [54].

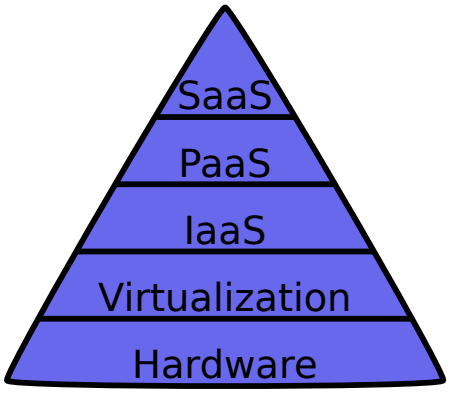
Soukromý cloud (angl. Private cloud) je obdoba veřejného cloudu, ovšem není dostupná široké veřejnosti, ale pouze v rámci firmy, která tento cloud provozuje, a jejich partnerů. Privátní cloud má stejné charakteristiky jako veřejný cloud — elasticitu a škálovatelnost. Hlavní rozdíl je ve způsobu řízení, kdy soukromý cloud řídí stejná firma, která jej používá, případně její partner, což umožňuje lépe přizpůsobit nabídku služeb reálným požadavkům.

Hybridní cloud (angl. Hybrid cloud) je spojení několika cloudů, které mohou být veřejné, soukromé nebo komunitní do jedné služby, jak je vidět na obrázku 3.1. Jednotlivé cloudy zůstávají jako samostatné entity, které jsou pomocí jednotného rozhraní spojeny do většího celku. Spojení může být buď trvalé nebo pouze na určitý čas, například pro splnění určitého úkolu [5]. Příkladem takového hybridního cloudu může být, když firma využívá vlastní datová úložiště pro uchovávání dat zákazníků a použije na zpracování výpočetní výkon, poskytovaný například službou EC2 od firmy Amazon [12].

Posledním typem cloudu je takzvaný *komunitní cloud*. Tento systém bývá vytvářen skupinou organizací, které mají například stejný obor zájmu nebo bezpečnostní politiku. Výhodou společné infrastruktury je rozdělení nákladů mezi více subjektů. Tento systém mohou řídit buď organizace sami nebo jiný poskytovatel. Příkladem takovéto služby je

Google Gov Cloud, což je nabídka firmy Google speciálně navržená pro americké vládní úřady.

Další způsob dělení je podle druhu služeb, které cloud poskytuje. Existuje několik různých způsobů rozdělení, zde bude popsáno dělení dle [24] na pět vrstev, jak je naznačeno na obrázku 3.2.



Obrázek 3.2: Rozdělení cloud computingu do vrstev podle poskytovaných služeb

V první vrstvě se nachází *Hardware*, někdy bývá tato vrstva označována jako HaaS (Hardware as a Service), kdy poskytovatel nabízí celé dedikované servery pro cloud computing. Jedná se vlastně o server housing, kdy i server je vlastnictvím poskytovatele housingu a ten jej pronajímá.

Další vrstva je *Virtualizace*, kdy nabízeným produktem je opět hardware, ale již virtualizovaný – tzn. nemusí se jednat o jedno fyzické zařízení, ale může jich být několik, které se budou pro zákazníka tvářit jako jedno nebo více fyzických zařízení. Do této skupiny by se dal zařadit produkt S3 od firmy Amazon. Jde o pronájem virtuálního disku včetně zálohování na serverech Amazonu za poplatek podle objemu uložených dat. Fyzicky se však nejedná o jeden disk, ale úložný prostor rozprostřený přes různé servery, který se navenek tváří jako fyzický disk.

Infrastructure as a Service (IaaS) umožňuje poskytování úložné a výpočetní kapacity podle požadavků a specifikací uživatele jako služby přes síť. Většinou se jedná o virtualizovaný operační systém, což umožňuje široké možnosti použití. Nejznámějším příkladem je služba Elastic Computing Cloud (EC2) firmy Amazon [15].

Dalším stupněm služeb je PaaS (*Platform as a Service*), což je podobně jako IaaS poskytnutí úložné a výpočetní kapacity na požádání. V tomto případě již nejde o univerzální řešení, ale prostředí pro běh určité aplikace nebo její části. Mezi nejznámější služby této kategorie patří Google AppEngine.

Poslední vrstva, SaaS (*Software as a Service*), je poskytnutí webové aplikace nebo služby přes Internet. Jedna instance dané služby běží v cloudu a obsluhuje velké množství uživatelů a organizací. Příkladem softwaru jako služby je GMail, Google Calendar nebo GitHub [37].

3.2 Technologie

Rozvoj cloud computingu vedl k rozvoji virtualizace jako nástroje, který zajišťuje lepší škálovatelnost výsledného produktu. Přehled nejpoužívanějších virtualizačních technologií uvedených v této kapitole je převzat z [13].

Virtualizační metody můžeme rozdělit do dvou kategorií: paravirtualizace a plná virtualizace [30]. Plná nebo také softwarová virtualizace pracuje tak, že *hypervisor* vytvoří virtuální hardware, ke kterému pak virtualizovaný systém přistupuje. Hypervisor je program, který řídí běh virtuálních strojů. Tato metoda umožňuje spustit libovolný operační systém bez nutnosti modifikace. Nevýhodou je degradace výkonu virtuálního stroje, protože každé volání hardwaru z virtualizovaného operačního systému musí být přeloženo na volání reálného hardwaru. Při paravirtualizaci hypervisor nabízí virtualizovanému systému rozhraní, které je podobné jako rozhraní pro přístup k reálnému hardwaru. Výhodou tohoto přístupu je, že odpadá nutnost odchytávat volání pro přístup hardwaru, proto je tato metoda rychlejší. Nevýhoda této metody spočívá v nutnosti provozování modifikovaných verzí operačních systémů ve virtuálním prostředí.

Pro zefektivnění běhu plně virtualizovaných systému je možné využít technologie Intel VT a AMD-V. V obou případech se jedná o *hardware-assisted virtualization*, čili podporu procesorů pro přímý přístup k hardwarovým zdrojům počítače z virtuálního stroje [45]. Jedná se tedy o nahrazení softwarového odchytávání a emulace instrukcí hardwarem.

Mezi zástupce nástrojů využívajících paravirtualizaci patří například *Xen*. Jeho hlavní výhoda je minimální zpomalení běhu virtualizovaných systémů. Nicméně je nutné, aby byl hostující systém specificky upraven pro běh v prostředí Xenu. Původně byl tento systém vyvinut na univerzitě v Cambridge, vedoucí jeho vývoje poté založil firmu XenSource, která poskytuje komerční podporu pro Xen. Tato společnost byla později koupena firmou Citrix. Xen také umožňuje migraci virtuálních systému za běhu [40]. Produkt EC2 firmy Amazon využívá Xen pro běh virtuálních strojů.

QEMU je emulátor různých typů procesorů, například x86, x86_64, PowerPC, ARM, MIPS a SPARC. QEMU také umožňuje emulovat jednotlivá zařízení, je možné jej tedy využit jako plnohodnotné virtualizační řešení. Jedná se o plnou virtualizaci [29]. QEMU podporuje velké množství hostitelských i hostujících systémů, například Linux, Solaris, Microsoft Windows a BSD.

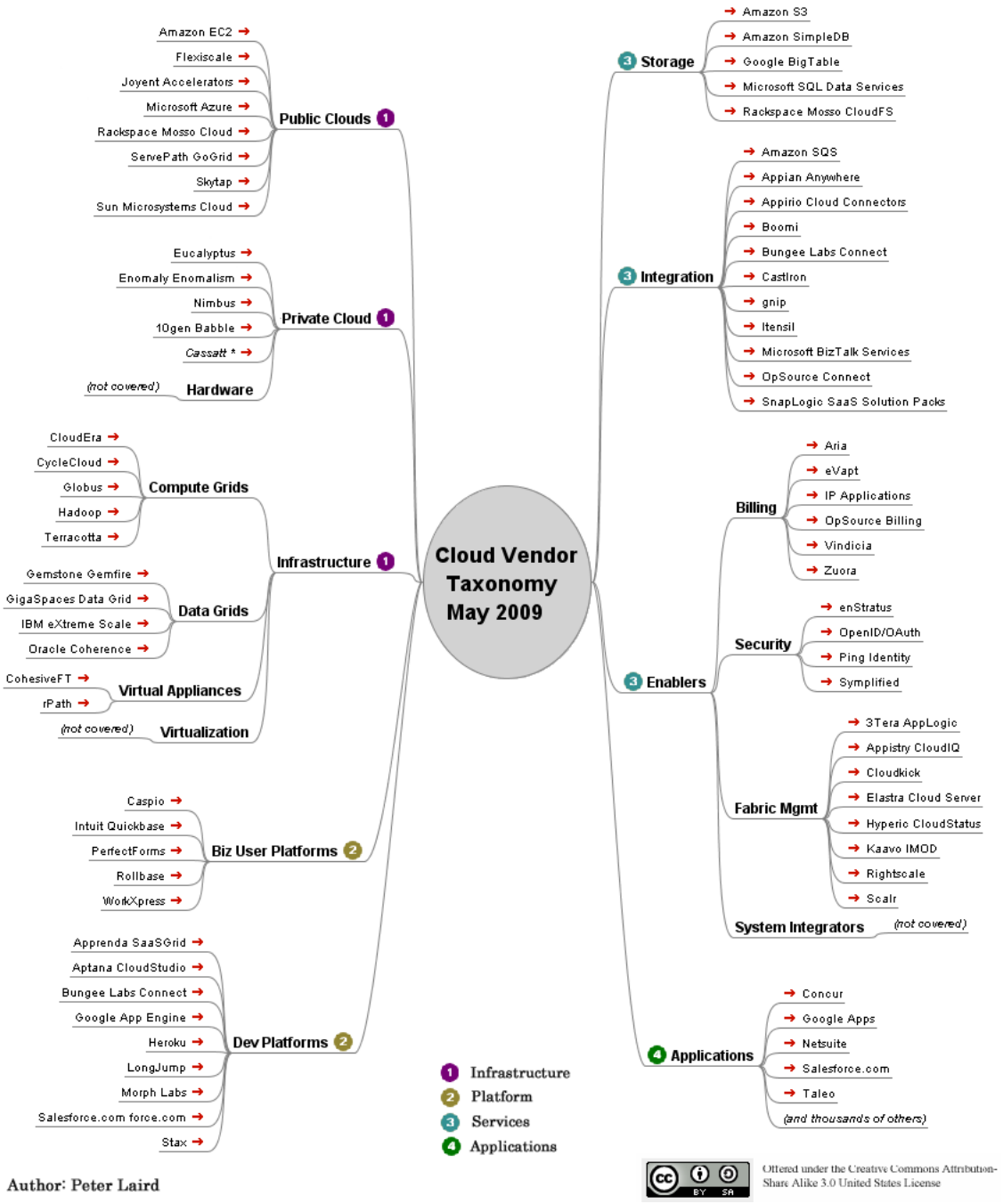
KVM je zkratka z Kernel-based Virtual Machine. KVM používá hardwarovou virtualizaci v procesoru pro běh virtuálního stroje přímo na hardwaru počítače bez nutnosti softwarového překladu volání. Pro emulaci zařízení jako displej, pevný disk, síťové karty a USB zařízení využívá QEMU [38].

3.3 Služby založené na cloud computingu

Obsahem této podkapitoly je popis nejznámějších produktů využívajících cloud computing.

Obrázek 3.3 znázorňuje variabilitu a množství služeb založených na cloud computingu. Cloudy jsou zde rozděleny do skupin podle toho, co zákazníkovi poskytují. Dělení na obrázku je mírně odlišné, než je uvedeno v této kapitole. Z tohoto obrázku je však patrné, že cloud computing nabízí mnoho firem a je možné jej využít k různým účelům. V první skupině jsou v tomto schéma označeny služby spadající do kategorie IaaS, pro přehlednost jsou zvlášť privátní a veřejné cloudy poskytující běh plnohodnotných virtuálních strojů a zbytek služeb poskytujících infrastrukturu.

Ve skupině 2 jsou služby z kategorie PaaS, rozdělené na služby pro vývoj aplikací a služby pro podporu podnikání. Třetí skupina stojí na pomezí mezi všemi ostatními kategoriemi. Jsou zde uvedeny služby poskytující úložný prostor (*Storage*), služby pro integraci (*Integration*) a služby pro zpřístupnění (*Enablers*), kam spadají systémy pro účtování, zabezpečení a management cloudů. Poslední kategorie obsahuje softwarové aplikace pracující v cloudu [23].



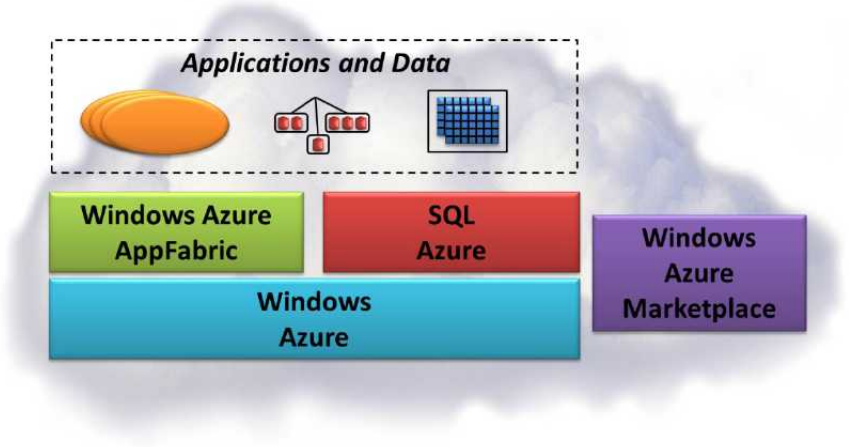
Obrázek 3.3: Mapa pojmů cloud computingu, převzato z [23].

Jeden z nejznámějších produktů spadajících do IaaS je Amazon EC2. Celý název této služby je *Amazon Elastic Compute Cloud*. Jedná se o službu poskytující virtuální stroje pro běh plnohodnotných operačních systémů. Výhodou je přidělování výpočetního výkonu na požádání. Ve webovém rozhraní si uživatelé této služby mohou nakonfigurovat parametry virtuálních strojů a dynamicky je měnit za běhu. Výsledná cena se odvíjí od skutečně využitého výpočetního času [49]. Amazon EC2 oficiálně podporuje operační systémy Linux

(RHEL, openSUSE, Fedora, Debian, CentOS, Gentoo a Oracle Linux), Microsoft Windows Server a OpenSolaris [48].

Google App Engine umožňuje vytvářet a hostit webové aplikace na infrastruktuře firmy Google. App Engine nabízí rychlý vývoj a uvedení do provozu, snadnou administraci bez nutnosti starat se o hardware a zálohování a další správu serverů [52]. Pro vývoj aplikací je možné použít několik programovacích jazyků. Preferované jsou jazyky Python a Java, ale je možné použít i další jazyky založené na interpretu JVM jako Groovy, JRuby, Scala, Clojure, Jython a další. Google App Engine patří do skupiny PaaS, což na rozdíl od IaaS neumožňuje použít libovolné prostředky pro vývoj, ale odpadá zde nutnost správy systému, na kterém aplikace běží.

Další poskytovatel PaaS je firma Microsoft se svým produktem Azure. Součástí Azure je operační systém pro webové služby, webová relační databáze a možnost propojení a spolupráce s .NET Services [6]. Platforma Azure zasahuje do různých typů cloud computingu (například Windows Azure patří do skupiny IaaS a Azure AppFabric je PaaS). Na obrázku 3.4 je rozdělení na jednotlivé služby.

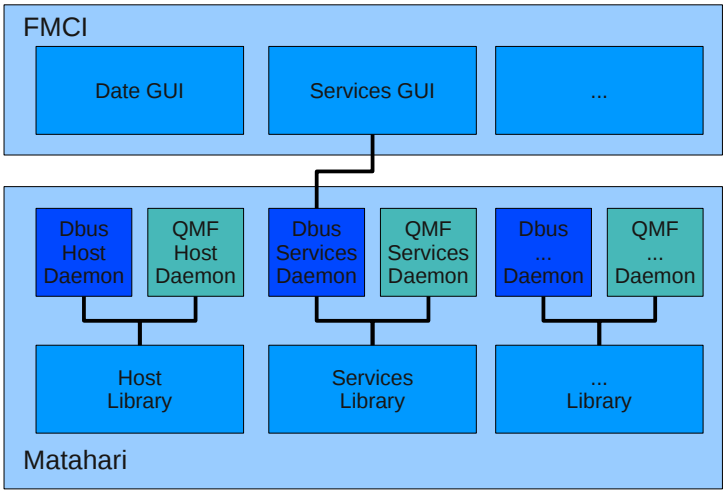


Obrázek 3.4: Rozdělení platformy Azure na jednotlivé produkty [6].

Kapitola 4

Architektura

Obsahem této kapitoly je návrh architektury systému pro správu. Při návrhu byla zohledněna snadnost údržby programů, které budou implementované rozhraní využívat. Pro zjednodušení je výsledné řešení rozděleno do dvou oddělených částí, jejichž vývoj probíhá samostatně. Vlastní konfigurace systému bude oddělena do sdílených knihoven. Nad nimi budou systémoví démoni, kteří budou rozhraní poskytovat ostatním aplikacím jak lokálně, tak i vzdáleně. Vývoj těchto knihoven stejně jako démonů probíhá v rámci projektu *Matahari*. Druhá část řešení pro správu systémů jsou programy s grafickým uživatelským rozhraním. Ty budou vyvinuty jako projekt *FMCI* – Fedora Management and Configuration Infrastructure. Vztah mezi rozhraním, které implementuje projekt Matahari, a nástroji pod hlavičkou FMCI je $M : N$. To znamená, že FMCI využije jen část poskytovaných rozhraní a použije i ostatní dostupná rozhraní. Případně budou vytvořena další rozhraní v rámci FMCI, pokud by nezapadala do koncepce projektu Matahari. Na obrázku 4.1 je znázorněn vztah mezi těmito dvěma projekty.



Obrázek 4.1: Schéma vztahu mezi projekty Matahari a FMCI.

Dále se tato kapitola zabývá systémem *Augeas*, který zjednodušuje přístup ke konfiguračním souborům a jejich editaci. Jedná se o obousměrnou transformaci mezi konfiguračním souborem a abstraktním stromem, který tyto soubory reprezentuje a umožňuje jejich

změnu. Projekt Matahari využívá Augeas ve svém konfiguračním agentovi, který byl vytvořen v rámci této práce.

4.1 Matahari

Cílem projektu *Matahari* [53] je poskytnout platformě nezávislé rozhraní pro konfiguraci systému sahající od monitorování volné paměti po zapínání a vypínání systémových služeb. Tyto služby jsou poskytovány aplikacemi, které se nazývají *agenti*. Termín agent je zde převzat z terminologie QMF a nejedná se o inteligentního agenta. Matahari poskytuje několik oddělených rozhraní, které jsou implementované jako sdílené knihovny. V současné době jsou podporovány systémy Microsoft Windows a Linux, konkrétně Fedora a RHEL. V plánu je však podpora pro další systémy.

Mezi tyto rozhraní patří *Host* pro zjišťování informací o systému, na kterém agent běží. Pomocí tohoto rozhraní lze zjistit například operační systém, celkovou operační paměť, volnou operační paměť, počet jader a procesorů a průměrné vytížení systému.

Další rozhraní *Network* poskytuje prostředky pro konfiguraci sítě. Protože komplexní podpora všech možných síťových řešení je nad rámec zaměření toho projektu, zaměřuje se Matahari pouze na na základní práci se sítí. Například je možné zapínat a vypínat síťová rozhraní a zjišťovat IP a MAC adresy.

Rozhraní *Services* se zaměřuje na služby, jejich zapínání, vypínání a kontrolu jejich stavu. Umožňuje také nastavit, zda se má služba spouštět automaticky po startu systému.

V rámci tohoto projektu bylo vytvořeno rozhraní *Config* pro čtení a editaci konfiguračních souborů, které používá systém Augeas pro práci s konfiguračními soubory jednotným způsobem.

Další rozhraní jsou zatím ve fázi plánování. Uvažuje se například o přidání možnosti přístupu k systémovému logu. Dále by bylo vhodné mít rozhraní pro instalaci a odstraňování programů. V linuxu je možné využít *PackageKit*, což je systém zapouzdřující balíčkovací systémy jednotlivých distribucí. Pro Windows však žádný balíčkovací systém neexistuje, bylo by tedy nutné stahovat instalátory jednotlivých programů a instalovat je automaticky. To představuje bezpečnostní riziko, protože neexistuje žádné ověřené úložiště instalačních balíků a stahováním z neověřených zdrojů se může do cílového počítače dostat škodlivý kód.

Dalším krokem ve vývoji projektu Matahari je integrace s existujícími nástroji pro správu unixových systémů. Například Spacewalk nebo Puppet mohou využít Matahari agenty jako rozhraní mezi spravujícím serverem a spravovanými klienty. Matahari agenti pouze vykonávají předem definované funkce, vnější logiku musí dodat právě nějaký specializovaný systém pro centralizovanou správu [32].

Matahari pro abstrakci rozdílů mezi jednotlivými platformami, na kterých běží, využívá projekt *Hyperic Sigar* [55]. Jedná se multiplatformní rozhraní pro přístup k nízkoúrovňovým informacím o hardwaru a operačním systému. Pro použití v rámci projektu Matahari je vhodná podpora většiny běžných operačních systémů. Sigar je implementovaný v jazyce C, je však možné použít jazyková rozhraní pro Javu, Perl, Ruby, Python, Erlang, PHP a C#.

4.2 FMCI

FMCI [58] je projekt, který má za cíl být centrálním bodem pro veškeré konfigurační nástroje ve Fedoře. To znamená, že se snaží sesbírat a sjednotit existující konfigurační

rozhraní a vytvořit nové, tam kde rozhraní chybí.

Tento projekt vznikl mimo jiné kvůli potřebě portu nástrojů system-config ve Fedoře na systém PolicyKit. Více o PolicyKitu je v kapitole 2. Port na tento systém není triviální, protože je nutno aplikaci rozdělit. Kód, který musí běžet se zvýšenými oprávněními, je nutné oddělit od uživatelského rozhraní. Další požadavek byl, aby kód pro konfiguraci programů a služeb použitelný i v jiných programech. Například spouštění a vypínání systémových služeb používají všechny nástroje pro konfigurace těchto služeb.

Proto bylo rozhodnuto, že bude použit projekt Matahari pro základní konfiguraci systému a služeb. Mimo to byly pod hlavičkou projektu *FMCI* sjednoceny existující nástroje pro správu systému a budou dopsány nové nástroje. Kvůli pročistění kódu stávajících nástrojů a sjednocení jejich grafického uživatelského rozhraní vznikla iniciativa *system config cleanup*. V první fázi byly existující nástroje prozkoumány a nalezené chyby byly nahlášeny do Bugzilly projektu Fedora. Dbalo se i na pročištění chyb v grafickém designu, přičemž se vycházelo z *GNOME Human Interface Guidelines* [1], což je seznam doporučení pro návrh grafických uživatelských rozhraní a zlepšení použitelnosti programů sepsaný pro potřeby projektu GNOME. Celkem bylo nahlášeno 189 chyb a návrhů na vylepšení [57]. K některým z těchto chybových hlášení byly také vytvořeny patche, které tyto chyby opravují. V současné době je opravena zhruba polovina těchto chyb. Zbytek chyb není z různých důvodů opraven. V některých případech je to kvůli časové náročnosti opravy, například u portů na systém PolicyKit, která vyžaduje značnou změnu v kódu programu. Některé z těchto nástrojů byly již nahrazeny novými nástroji (*system-config-securitylevel* byl nahrazen nástrojem *system-config-firewall*) nebo zrušeny bez náhrady. Například *system-config-soundcard* již nebyl třeba, protože jeho funkci převzaly konfigurační nástroje desktopových prostředí.

4.3 Augeas

Běžným způsobem konfigurace většiny nástrojů v unixových systémech je editace konfiguračních souborů. Tato operace je klíčová pro většinu nástrojů pro nastavení systému, proto je vhodné tento úkon nějakým způsobem automatizovat, případně jej zjednodušit. Obecně se jedná o netriviální úkol, protože existuje velké množství různých formátů konfiguračních souborů. Všechny nástroje by tedy musely nějakým způsobem implementovat přečtení konfiguračního souboru, jeho prezentaci uživateli, zpracování požadovaných změn a opětovný zápis do souboru. U zápisu je většinou požadováno, aby konfigurační soubor co nejméně změnil, čili zůstaly zachovány komentáře i formátování souboru. Velmi výrazným zjednodušením je použití systému *Augeas*, který bude v této podkapitole popsán. Informace jsou čerpány z oficiálních stránek projektu Augeas [59] a ze stránek projektu *Boomerang* [61], z něhož Augeas vychází. Cílem projektu Boomerang (dříve Harmony) je vytvořit obousměrný (angl. bidirectional) programovací jazyk pro popis transformací mezi různými formáty textových dat.

Augeas je nástroj pro editaci konfiguračních souborů. Pracuje na principu transformace mezi konfiguračním souborem a stromem, který jej reprezentuje. Na tomto stromě provede zadané úpravy a poté transformuje tento strom zpět na konfigurační soubor. Pro popis pravidel, kterými se transformace na strom řídí, slouží tzv. *lenses*. Jedná se o pravidla napsaná v podmnožině funkcionálního jazyka ML. Pro každý konfigurační soubor je nutné vytvořil tento *lens*, který poté mohou využívat všechny programy pro editaci tohoto souboru. Výhodou je, že konfigurační soubor zůstane zachován v původní podobě včetně formátování a komentářů a změní se pouze explicitně editované části.

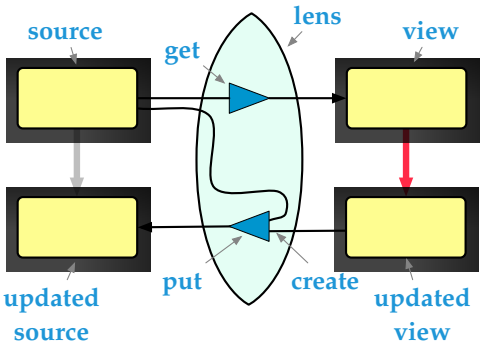
Součástí projektu Augeas je shellový skript `augtool`, pomocí kterého je možné provádět libovolné operace nad vytvořenými stromy. Mimo základního rozhraní v jazyce C je možné také využít rozhrání pro další programovací jazyky jako Ruby, Python, OCaml, Perl, Haskell, PHP a Javu.

Každý uzel stromu vytvořeného transformací z konfiguračního souboru obsahuje tři položky. Popisek (label) a hodnota (value) jsou textové řetězce. Popisek slouží k identifikaci daného uzlu při jeho procházení. Dále uzel obsahuje seznam všech potomků daného uzlu.

Přístup k jednotlivým uzlům stromu se provádí pomocí výrazů, které jsou inspirovány jazykem XPath. Prefixem takovéto cesty k uzlu je identifikace souboru: `/files/etc/hosts`. Dále jsou popisky jednotlivých uzlů oddělené lomítkem. Na rozdíl od cesty v souborovém systému je možné, aby sourozenecké uzly měly stejné popisky. Potom se k jednotlivým uzlů přistupuje pomocí indexace. Například druhý alias prvního záznamu v souboru `/etc/hosts` je možné získat pomocí výrazu `/files/etc/hosts/1/alias[2]`. Také je možné přistoupit k poslednímu prvku pomocí `alias[last()]`. Pro prohledávání stromu je možné také použít divokou kartu (angl. wildcard) `*`. Cestě `p/*/g` vyhovují všechny uzly s popiskem `g`, které mají prarodičovský uzel `g`. Takovéto vyhledávání není rekurzívní, čili výše uvedenému výrazu neodpovídá uzel s cestou `p/a/b/g` [27].

Základní operace

Všechny zde popsané příkazy se vztahují pro použití se skriptem `augtool`, použití přímo z programovacích jazyku je obdobné. Na obrázku 4.2 je nákres, jak pracují některé příkazy.



Obrázek 4.2: Terminologie *lens* souborů [61].

První skupinou příkazu jsou příkazy pro procházení stromem a zjišťování hodnot uzlů. Pro získání hodnoty, která náleží určitému uzlu, je možné použít příkaz `get`, jehož parametrem je cesta k uzlu. Pro zjištění potomků daného uzlu složí příkaz `ls` opět s cestou k uzlu jako parametrem. Příkaz `print` vypíše celý podstrom začínající cestou, která je nepovinným parametrem tohoto příkazu. Implicitně se vypisuje celý strom.

Pro transformaci stromu slouží několik dalších operací, které umožňují měnit strukturu stromu a hodnoty jednotlivých uzlů. Příkaz `set` nastaví hodnotu uzlu definovaného cestou v prvním parametru příkazu na hodnotu ve druhém parametru. Pokud uzel odpovídající takovéto cestě neexistuje, pak bude vytvořen včetně všech neexistujících rodičovských uzlů. Všechny přidané uzly budou jako poslední potomci rodičovského uzlu. Pokud je požadováno umístění uzlu na určitou pozici ve stromě, je nutné použít operaci `ins`. Tato operace má tři parametry: první je popisek vkládaného uzlu, druhý určuje, zda se má uzel vytvořit před nebo za uzel, jehož cesta je zadána třetím parametrem.

Posledním klíčovým příkazem je `save`, který slouží pro uložení upraveného stromu zpět do souboru. Operace, která bude po zavolání tohoto příkazu provedena, lze změnit nastavením hodnoty uzlu s cestou `/augeas/save`. Možné hodnoty jsou `noop`, `backup`, `newfile` a `overwrite`. Po nastavení tohoto uzlu na `noop` se neprovede při uložení žádná operace, což je možné využít při hledání chyb, které mohou nastat při uložení. `Backup` zálohuje původní soubor tak, že jej před uložením přejmenuje přidáním koncovky `.augsave`. Použití `newfile` znamená uložení modifikovaného souboru s příponou `.augnew`, takže původní soubor zůstane zachován. Poslední hodnota slouží k přepsání původního obsahu novým bez zálohy.

Lens

Pro svoji funkci potřebuje Augeas jak popis transformace z konfiguračního souboru na strom, tak i opačně. Vytvářet dvě mapování odděleně by bylo nejen neefektivní z hlediska vynaložené práce, ale mohlo by to vést k nekompatibilitám mezi těmito dvěma transformacemi. Proto Augeas využívá obousměrné mapování, kdy stačí pouze jeden soubor pro popis obou směrů transformace. Také je nutné zachovávat informace ze zdrojového souborů, které nebudou v abstraktním stromě obsaženy (např. formátování). Nejedná se tudíž o bijektivní zobrazení, protože konfigurační soubory s různým formátováním se mohou zobrazit na stejný strom.

Formálně se lens skládá ze dvou funkcí, *get* a *put*. Nechť C je množina všech konkrétních datových struktur (v tomto případě řetězců) a A je množina všech abstraktních datových struktur (stromů), lens l se skládá z operací 4.1 a 4.2.

$$l.get \quad : \quad C \rightarrow A \tag{4.1}$$

$$l.put \quad : \quad A \times C \rightarrow C \tag{4.2}$$

Funkce *put* bere jako první parametr abstraktní struktury, které byly získána operací *get*, a jako druhý parametr původní konkrétní struktury. Ty jsou poté použity pro obnovení informací, které byly operací *get* vynechány (typicky formátování). Lens l musí splňovat pro každé $c \in C$ a každé $a \in A$ rovnice 4.3 a 4.4 [27, 11].

$$l.put\ (l.get\ c)\ c \quad = \quad c \tag{4.3}$$

$$l.get\ (l.put\ a\ c) \quad = \quad a \tag{4.4}$$

Tyto dvě pravidla umožňují abstrahovat nepodstatné detaily při vytváření abstraktního stromu, ale zpětně je využít při transformaci do konfiguračního souboru. Pokud by byly tyto operace definovány jako inverzní, musel by být mezi konkrétním a abstraktním pohledem bijektivní vztah. To by znamenalo, že i informace o odsazovaní, komentáře a ostatní nepodstatné informace musely být součástí abstraktního stromu, což je nežádoucí.

Lens soubory se vytváří pomocí jednoduchých vestavěných pravidel, takzvaných lens primitiv, a jejich kombinací pomocí len kombinátorů. Obvykle jeden komplexní lens soubor, který zpracovává jeden konfigurační soubor, je složen z několika lens pravidel, která zpracovávají určité části souboru. Na nejnižší úrovni jsou všechna pravidla sestavena z lens primitiv pomocí základních kombinátorů.

Lens primitiva

Lens primitiva jsou ve své podstatě funkce, které ze svých vstupních parametrů vytváří lens pravidlo, čili popis transformace mezi konkrétní a abstraktním pohledem a zpět. Parametry

těchto funkcí jsou buď textové řetězce, které zde budou označeny `STR`, nebo regulární výrazy označeny jako `RE`.

Pro nastavení popisku uzlu složí primitivum `key RE`, které při čtení uloží do popisku text odpovídající danému regulárnímu výrazu. Při zápisu tuto uloženou hodnotu zapíše na původní místo do souboru. Pokud je třeba uložit do popisku text, který není součástí vstupního souboru, je možné použít `label STR`. Tento řetězec nebude ani zapsán do výstupního souboru.

Podobně jako `key` nastavuje popisek, `store RE` ukládá hodnotu odpovídající regulárnímu výrazu jako hodnotu daného uzlu. Pokud se ukládá do hodnoty uzlu text, který není součástí vstupního souboru, použije se `value STR`.

`seq STR` je podobné jako `label`. Při operaci *get* nastavuje popisek uzlu na aktuální číslo sekvence označené daným parametrem. Tento příkaz slouží k vytváření sourozeneckých uzlů, které by měly stejný popisek. Při operaci *set* nehraje hodnota tohoto uzlu roli, pouze se kontroluje, zda je nastavena na nezápornou hodnotu.

Posledním primitivem je `del RE STR`. Při čtení vstupního souboru je hodnota odpovídající regulárnímu výrazu `RE` ignorována a není zapsána do stromu. Při výstupu se vezme hodnota ze vstupního souboru a ta se zapíše. Toto primitivum je použito pro zpracování formátování, kde prázdné znaky nejsou transformovány na strom, ale při zápisu zůstávají zachovány. Parametr `STR` je použit pouze při vytvoření uzlu.

Lens kombinátory

Kombinátory slouží pro vytváření složitějších pravidel z výše uvedených primitiv. V následujícím odstavcích budou pomocí l, l_1 a l_2 označeny libovolná lens pravidla.

První z kombinátorů je vytvoření podstromu. Syntakticky se zapisuje takto: `[` l `]`. Lens l bude vložen jako potomek současného uzlu. Pokud je l strom, bude vložen jako podstrom k současnému uzlu. Další kombinátor je konkatenace, zapsána l_1 `.` l_2, slouží k postupné aplikaci jednotlivých lens. Na vstupu je nejprve provedeno pravidlo l_1, poté l_2. Kombinátor sjednocení, l_1 `|` l_2, vybírá, jestli použít l_1 nebo l_2. Vybere je pravidlo, které je možné na vstupní text při operace *get*, resp. strom při operaci *put*, aplikovat. Poslední kombinátor je iterace. Je možné použít dvě verze tohoto kombinátoru: l^* a l^+. První z nich znamená libovolně opakovaní lensu l, druhá vyžaduje alespoň jedno opakování.

Kapitola 5

Implementace

Tato kapitola se zabývá implementací navrženého systému. Je zde popsáno, jak jsou implementovány některé části systému popsáné v předchozí kapitole. Projekt Matahari poskytuje jak QMF, tak i D-Bus rozhraní. Bylo tedy vhodné maximálně automatizovat vytvoření rozhraní pro obě tyto varianty. Nejprve je nutné mít popis poskytovaných rozhraní, který bude definovat rozhraní celého systému a jeho rozčlenění. Byl vybrán definiční soubor pro QMF, který je ve formátu XML. Z tohoto souboru je automaticky generováno rozhraní pro systém D-Bus, které je taktéž definováno pomocí XML souboru. Pro zjednodušení se generuje i část kódu v jazyce C, která implementuje dané rozhraní. Bohužel není možné automaticky generovat celý kód, protože je nutné manuálně převést funkce a vlastnosti D-Bus rozhraní na funkce v příslušné knihovně. Dalším krokem je integrace PolicyKitu. PolicyKit zajišťuje, že pouze uživatel s patřičnými oprávněními bude moci využít vytvořené rozhraní. Pro zjednodušení implementace byla použita knihovna GLib, která obsahuje pokročilé datové typy, objektový systém GObject a další prostředky, které usnadňují vývoj. Pro implementaci D-Bus rozhraní byla použita knihovna dbus-glib, která poskytuje rozhraní mezi systémem D-Bus a knihovnou GLib.

V rámci této diplomové práce byla do projektu Matahari přidána podpora pro rozhraní D-Bus, systém překladu ze zdrojových kódu byl modifikován tak, aby byl modulární a snadno rozšiřitelný. S tím souvisí přidání podpory pro samostatné Matahari agenty, které pouze využijí vytvořenou infrastrukturu, avšak budou distribuováni samostatně. Dále byl vytvořen samostatný *config* agent, který pomocí Augeasu umožňuje číst a editovat konfigurační soubory. Poté byl vytvořen testovací nástroj, který umožňuje porovnat výstupy z jednotlivých částí systému. Tento nástroj je popsán v kapitole testování.

5.1 Knihovny

Při návrhu knihoven bylo dbáno na multiplatformnost a snadné přidání podpory pro další platformy, na kterých Matahari poběží. V této podkapitole je ukázán příklad návrhu knihovny *host* pro zjišťování informací o systému, na němž Matahari běží. Rozhraní knihovny je definováno v souboru `host.h`, kde jsou uvedeny deklarace všech funkcí jako externí symboly pomocí klíčového slova jazyka C `extern`. Tento hlavičkový soubor je poté vložen do agenta, který tyto funkce volá. Definice těchto funkcí jsou v souboru `host.c`. Pokud je dostupná multiplatformní implementace dané funkce, je v tomto souboru. Pokud je daná funkce závislá na operačním systému, je v této funkcí zavolána odpovídající platformě závislá funkce. Například ve funkci `host_reboot` pro restartování hostitelského systému je

volána funkce `host_os_reboot`. Podle operačního systému, pro který je systém překládán, se přeloží zdrojový soubor, kde je tato funkce definována. Tedy pro linux je přeložen soubor `host_linux.c`, ve kterém funkce `host_os_reboot` volá funkci `reboot` ze standardní knihovny jazyka C s parametrem `LINUX_REBOOT_CMD_RESTART`. Tento parametr nemá alternativu pro systém Windows, proto v souboru `host_windows.c` je tato funkce implementována voláním WinAPI funkce `ExitWindowsEx`. Při překladu se určí, které soubory se mají překládat podle zvolené platformy. Při přidání podpory pro další platformu se pouze vytvoří nový zdrojový soubor, kde se tyto funkce implementují a upraví se překlad. Nevýhoda tohoto řešení je nutnost zduplikovat kód, pokud by některé platformy měli určité funkce stejné. Například při přidání podpory pro BSD systémy by byla odlišná funkce `host_reboot`, ale ostatní funkce stejné jako pro linux. Nicméně buď by se musely tyto funkce zkopírovat do `host_bsd.c` nebo přidat společný soubor pro tyto dvě platformy a opravit překlad.

5.2 Generování QMF a D-Bus rozhraní

Protože je nutné mít stejné QMF i D-Bus rozhraní, bylo by zbytečné psát oba definiční soubory zvlášť a po každé změně upravovat oba. Lepší způsob je napsání pouze jednoho souboru a automatické generování druhého. Jak QMF, tak i D-Bus mají popsané své rozhraní pomocí XML souboru, takže lze použít XSL transformaci.

Základní rozdělení XML souboru s definicí QMF rozhraní je v příkladu 5.1. DTD soubor s definicí QMF rozhraní sice neexistuje, jeho struktura je však jednoduchá a dobře popsaná na stránkách QMF [46]. Jedná se o část objektu pro přístup k systémovým službám. Název této třídy je *Services*, má jednu vlastnost s názvem *hostname*, která je typu short string (řetězec do 255 znaků) a je pouze pro čtení. Atribut index znamená, že tato vlastnost slouží pro unikátní identifikaci objektu. Tento atribut může mít i více vlastností. Dále tato třída obsahuje jednu metodu s názvem *list*. Atribut *desc* obsahuje slovní popis této metody. Tato metoda má jeden argument, který se jmenuje *services* a je typu seznam. dir="O" znamená, že se jedná o výstupní parametr.

Příklad 5.1: Ukázka definice QMF rozhraní

```
1 <schema package="com.redhat.matahari">
2   <class name="Services">
3
4     <property name="hostname" type="sstr" access="RO" index="y"/>
5     <method name="list" desc="List known system services">
6       <arg name="services" dir="O"  type="list" />
7     </method>
8   </class>
9 </schema>
```

Po transformaci by měl rozhraní pro systém D-Bus vypadat tak, jak v příkladu 5.2. Výsledný XML soubor musí odpovídat DTD definici [44] pro D-Bus introspekci. Základním elementem je *node*, který může mít nepovinný atribut *name* obsahující cestu k objektu. Každý *node* obsahuje několik elementů *interface*, které mají povinný atribut *name* — název rozhraní. Dále následuje popis vlastností (*property*) a metod spolu s jejich argumenty, které poskytuje dané rozhraní. Řádek 3 v tomto příkladu obsahuje anotaci, která je použita pro automatické generování hlavičkových souborů. Anotace poskytuje dodatečné informace, které nejsou nutné pro systém D-Bus, ale pro další podpůrné nástroje, například `dbus-binding-tool`, který je popsán později. Existuje několik typů anotací, přičemž jejich typ je určen atributem *name*. Anotace `org.freedesktop.DBus.GLib.CSymbol` slouží

ke spojení D-Bus rozhraní s objektem typu GObject v jazyce C [36]. Název tohoto objektu musí být shodný s hodnotou atributu *value*. Klíčové je označení všech metod anotací `org.freedesktop.DBus.GLib.Async`. Všechny takto označené metody budou asynchronní a bude jim přidán parametr typu ukazatel na `DBusGMethodInvocation`. Pomocí tohoto parametru je možné určit kontext volání funkce, což je klíčové pro autorizaci metod pomocí systému PolicyKit, který bude popsán dále.

Příklad 5.2: XML soubor popisující D-Bus rozhraní

```
<node>
  <interface name="com.redhat.matahari.Services">
    <annotation name="org.freedesktop.DBus.GLib.CSymbol" value="Services"/>
    <property name="hostname" type="s" access="read"/>
    <method name="list">
      <arg name="services" direction="out" type="as"/>
    </method>
  </interface>
</node>
```

XSL znamená eXtensible Stylesheet Language a XSL transformace slouží k automatickému převodu mezi XML soubory. Oba XML soubory mají velmi podobnou strukturu, často se mění jen název elementu. Jak je vidět z příkladu 5.3, prvním krokem je nahrazení kořenového elementu *scheme* elementem *node*. Na řádcích 3–5 je uložení názvu balíčku do proměnné *package* pro snadnější přístup. Pomocí konstrukce *for-each*, která slouží pro iteraci přes všechny elementy odpovídající danému XPath výrazu v parametru *select*, se vytvoří z každé třídy v QMF jedno D-Bus rozhraní. Název tohoto rozhraní je konkatenace názvu balíčku (proměnná *package*), tečky a názvu třídy (parametr *name* elementu *class*).

Příklad 5.3: Převod základní struktury QMF rozhraní na D-Bus rozhraní

```
1  <xsl:template match="/">
2      <node>
3          <xsl:variable name="package">
4              <xsl:value-of select="schema/@package" />
5          </xsl:variable>
6          <xsl:for-each select="schema/class">
7              <interface>
8                  <xsl:attribute name="name">
9                      <xsl:value-of select="concat($package, '.', @name)" />
10                 </xsl:attribute>
11                 <!-- Zde nasleduje zpracovani vlastnosti a metod -->
12             </interface>
13         </xsl:for-each>
14     </node>
15 </xsl:template>
```

Zpracování vlastností a statistik je uvedeno v příkladu 5.4. D-Bus používá pro všechny atributy typ *property*, ale QMF je dělí na *property* a *statistics*. Pokud se vlastnost objektu nemění po celou dobu běhu programu, používá se typ *property*. Pro dynamicky se měnící atributy (např. čítače) je použit typ *statistics*. Dále je potřeba převést typy těchto atributů z typového systému QMF na systém používaný v definičních souborech D-Busu. Protože je nutné tuto operaci provést i pro parametry metod, je to provedeno v šabloně *type*, která je volána pomocí konstrukce *call-template*. Stejně tak je šablonou zkonvertován parametr *access*, který slouží pro informací, zda je o vlastnost, která je zapisovatelná nebo pouze pro čtení.

Příklad 5.4: Zpracování vlastností a statistik QMF modelu

```
<xsl:for-each select="property|statistic">
    <property>
        <xsl:attribute name="name">
            <xsl:value-of select="@name" />
        </xsl:attribute>
        <xsl:call-template name="type" />
        <xsl:call-template name="access" />
    </property>
</xsl:for-each>
```

Obdobně se zpracují i metody, jak je vidět v příkladu 5.5. Každá metoda může mít libovolný počet parametrů, které jsou popsány pomocí elementů *arg*. Tyto parametry mají obvykle tři atributy: název, datový typ a určení směru, zda je o parametr vstupní nebo výstupní. Pro transformaci jsou opět použity šablony.

Příklad 5.5: Transformace popisu metod mezi QMF a D-Bus

```
<xsl:for-each select="method">
    <method>
        <xsl:attribute name="name">
            <xsl:value-of select="@name" />
        </xsl:attribute>
        <xsl:for-each select="arg">
            <arg>
            <xsl:attribute name="name">
                <xsl:value-of select="@name" />
            </xsl:attribute>
            <xsl:call-template name="type" />
            <xsl:call-template name="dir" />
            </arg>
        </xsl:for-each>
    </method>
</xsl:for-each>
```

Šablona pro převedení směru argumentu je triviální, protože se jedná o pouhé nahrazení $I \rightarrow in$ a $O \rightarrow out$. Šablona pro přístupová práva k vlastnosti objektu je podobně jednoduchá, ovšem QMF má na rozdíl od D-Busu možnost mít parametr s právem *read-create*, čili práva pro čtení, zápis i vytvoření. D-Bus neumožňuje mít nevytvořenou vlastnost, takže je dostačující nahradit toto právo právem RW – čtení a zápis.

Nejkomplikovanější je šablona pro převod datových typů parametrů. Základní datové typy jako celá čísla, čísla s plovoucí řádovou čárkou, boolovské hodnoty a řetězce mají své odpovídají protějšky, avšak strukturované typy nelze jednoduchým způsobem převést. Pro účely tohoto projektu bylo nutné alespoň nějak převést typ seznam. QMF ve svých definičních souborech nerozlišuje mezi typem prvků seznamu. Naopak D-Bus potřebuje vědět typ dat uložených v seznamu. Pro univerzální automatický převod mezi definičními soubory by bylo nutné tuto informaci nějak uložit do definice QMF rozhraní, nicméně neexistuje standardizovaný způsob. Bylo by možné použít například komentáře ve zdrojovém XML souboru, které je možné pomocí XSL transformace číst. Nicméně pro potřeby tohoto projektu je dostačující automaticky předpokládat, že se jedná o seznam řetězců. Obdobný problém nastává s datovým typem slovník, který je v QMF označen jako `dict`, nicméně DBus vyžaduje znát jak datový typ klíče, tak hodnoty. Zde byl zvolen odlišný přístup. Datový typ klíče je vždy řetězec, ale hodnota bude určena dynamicky za běhu programu. Jako vhodné pro tento problém se ukázalo řešení, kdy se při implementaci Matahari agenta

vytvoří funkce s názvem `matahari_dict_type`, která jako parametr požaduje identifikaci vlastnosti objektu. Tato funkce vrací typ hodnot ve slovníku pro tuto vlastnost. Vynucení implementace této funkce je dosaženo uvedením jejího prototypu v hlavičkovém souboru, který Matahari agenti vyžadují. Pokud by tělo této funkce nebylo uvedeno, linker zahlásí chybu při sestavování programu.

5.3 Implementace QMF agentů

Při implementaci QMF agentů je použit jazyk C++ kvůli využití objektově orientovaného paradigma. Základem všech agentů je třída `MatahariAgent`, která obsahuje metody pro vytvoření a spuštění agenta. Každý agent musí tuto třídu zdědit a reimplementací virtuálních tříd určit svoje chování. V metodě `setup` je možné nastavit vlastnosti objektu QMF rozhraní. Metoda `invoke` je vykonána při většině akcí jako dotazu na agenta, jeho subskripci ke QMF brokeru či volání metody agenta. Právě na volání metody bude zpravidla agent reagovat, ostatní akce lze ignorovat, protože jsou ošetřeny implicitně. Součástí akce volání metody je název metody a její parametry, které je však nutné převést z datových typů systému QMF na běžné datové typy jazyka C a knihovny GLib. Obdobně je třeba provést transformaci datových typů pro návratové hodnoty.

5.4 Generování kódu pro D-Bus

Dalším krokem k výslednému řešení bylo generování části kódu, která zjednoduší implementaci rozhraní pro D-Bus. Tento krok se skládá ze dvou nezávislých částí. První z nich je zpracování metod, druhá se zabývá vlastnostmi objektů.

Při zpracování volání metody přes systém D-Bus je nutné zvolit odpovídající funkci v jazyce C a převést vstupní i výstupní parametry z typů systému D-Bus na typy, se kterými je možné pracovat v jazyce C (v tomto případě datové typy poskytované knihovnou GLib). O to se stará program `dbus-binding-tool`, jenž je součástí distribuce knihovny `dbus-glib`. Tento nástroj z XML popisu D-Bus rozhraní vytvoří hlavičkový soubor jazyka C, jenž obsahuje definice funkcí poskytovaných rozhraním. Dále tento program transformuje parametry těchto metod.

Zpracování vlastností objektů systému D-Bus je opět automatizováno tak, aby jeho použití bylo pro vývojáře snazší. Pomocí XSL transformace se z XML popisu D-Bus rozhraní vytvoří hlavičkový soubor jazyka C, který obsahuje výčet všech použitých vlastností pro dané rozhraní a pole záznamů, které k dané vlastnosti přiřazuje název, popis, přístupové právo a příznak datového typu. Tento příznak je později převeden na datový typ z knihovny GLib.

Oba tyto hlavičkové soubory jsou později vloženy do zdrojového souboru, který musí implementovat prototypy daných funkcí. Celý přístup k jednomu objektu rozhraní D-Bus je zapouzdřen do objektu typu GObject, což je implementace objektového orientovaného paradigma v jazyce C. Při inicializaci tohoto objektu jsou vytvořeny vlastnosti odpovídající vlastnostem v D-Bus rozhraní. K tomu se využije pole záznamů popsané výše, musí se však transformovat typy těchto vlastností. Podle příznaku typu ze záznamu vlastnosti je vytvořen odpovídající datový typ knihovny GLib.

5.5 Řízení přístupu pomocí PolicyKitu

Matahari agenti potřebují běžet se zvýšenými oprávněními, proto je nutné řídit přístup uživatelů k jednotlivým funkcím a vlastnostem. Systém QMF obsahuje vlastní řešení uživatelských oprávnění, kdy pro autentizaci je možné použít SASL a Kerberos [42]. Autorizace uživatelů pro jednotlivé akce je možné ovlivnit nastavením ACL (Access Control List) pravidel.

Pro řízení přístupu pro D-Bus rozhraní byt zvolen PolicyKit. Všechny metody a vlastnosti mají definovánu akci, která určuje nastavení oprávnění uživatelů pro tuto akci. Pro zjednodušení vytváření dalších agentů je součástí projektu Matahari XSL soubor, který z popisu D-Bus rozhraní vygeneruje soubor práv pro PolicyKit. Tento skript automaticky nastavuje nutnost autorizace jako uživatel se zvýšenými oprávněními pro všechny metody a vlastnosti, které mají povolen zápis. Pro vlastnosti bez práva zápisu není nutné žádné ověření autorizace, každý může tyto vlastnosti číst. Toto nastavení se však vztahuje pouze na výchozí stav po vygenerování akcí a je jej možné kdykoli libovolně změnit.

Při implementaci všech metod je nutné ověřit autorizaci uživatele bez ohledu na výchozí nastavení autorizace dané akce, protože administrátor systému, kde Matahari poběží, může toto nastavení změnit. K tomuto ověření slouží metoda `polkit_authority_check_authorization`, která vyžaduje jako jeden ze svých parametrů instanci objektu `PolkitSubject`. Tento parametr určuje, v jakém kontextu je program spuštěn. Kontext obsahuje mimo jiné informace o uživateli, který danou akci požaduje. Pokud je potřeba ověřit heslem, zda je uživatel pro danou akci autorizován, informace z tohoto kontextu určí, kterému z přihlášených uživatelů se dialog pro zadaní hesla zobrazí. Tento kontext je možné získat z parametru typu `DBusGMethodInvocation` pomocí metody `dbus_g_method_get_sender`, která vrací identifikaci volajícího. Tento parametr je přítomen u všech asynchronních funkcí.

Pro autorizaci přístupu k vlastnostem je nutné při každé čtení i zápisu ověřit práva uživatele pro danou akci. Toho je dosaženo redefinicí metod `Get` a `Set` z rozhraní `org.freedesktop.DBus.Properties`, což je standardní rozhraní pro přístup k vlastnostem objektu. V obsluze volání těchto dvou metod se ověří autorizace uživatele a poté se přečte nebo zapíše vlastnost odpovídajícímu objektu typu GObject.

5.6 Obecný konfigurační agent

Při portování system-config nástrojů na systém PolicyKit se ukázalo, že většina nástrojů pouze potřebuje se zvýšenými oprávněními upravovat konfigurační soubory. U většiny těchto nástrojů tedy port na PolicyKit spočíval ve vytvoření jednoduchého programu (tzv. *helper*), který byl spuštěn D-Bus démonem se zvýšenými oprávněními. Tento program pouze na vyžádání udělal patřičné změny konfiguračních souborů. Pro správnou funkci tohoto programu je nutné vytvořit jednak XML popis poskytovaného rozhraní, dále konfigurační soubory pro D-Bus. Konkrétně je třeba soubor v `/usr/share/dbus-1/system-services`, který umožní automatickou aktivaci na požádání a svazuje cestu ke spustitelnému souboru poskytujícím D-Bus rozhraní s názvem tohoto rozhraní. Pro nastavení, pod jakým uživatelem může dané rozhraní běžet a kdo k němu smí přistupovat, slouží soubor v `/etc/dbus-1/system.d`. Mimo to je třeba ještě vytvořit konfigurační soubor po PolicyKit, který bude obsahovat názvy akcí a výchozí nastavení oprávnění pro tyto akce. Tyto kroky jsou stejné u většiny grafických rozhraní pro nastavování systému, proto by bylo vhodné je nějak minimalizovat.

Jako univerzální řešení se jeví vytvoření Matahari agenta, který by zprostředkovat rozhraní systému Augeas přes rozhraní D-Bus. Konfigurační nástroje by poté pro změnu konfiguračního souboru zavolaly tohoto agenta, který by ověřil jejich autorizaci a pomocí Augeasu upravil daný konfigurační soubor. Zjednodušení portování aplikací na PolicyKit by bylo značné, protože by nebylo nutné vytvářet helper, tím by odpadly i kroky pro vytvoření konfigurace pro D-Bus.

Při implementaci tohoto agenta však nastal problém v tom, že funkce Augeasu nejsou bezestavové. Je tedy nutné, aby byla nejprve zavolána funkce `aug_init`, která vrací ukazatel na strukturu `augeas`. Tento ukazatel obsahuje informace o stromu odpovídajícím všem známým konfiguračním souborům a změnách v nich. Proto musí tento ukazatel být předán všem funkcím jako parametr. Teoreticky by bylo nejvhodnější tuto strukturu po každé operaci serializovat a vrátit volajícímu programu, nicméně kromě obtížné technické realizace je toto řešení také dost náročné výpočetně a značně by zpomalilo celý běh programu. Bylo zvoleno řešení spočívající v uložení struktury `augeas` do pole a vrácení indexu volajícímu programu. Tento program musí nejprve zavolat inicializační funkci, která vrátí index stromu s provedenými změnami. Tento index musí být poté předán jako parametr všem volaným funkcím. Po ukončení práce by měla být zavolána funkce `close`, která uvolní strukturu `augeas` z paměti. Pokud však klientská aplikace tuto funkci nezavolá, zůstane tato struktura v paměti po celou dobu běhu programu. Vzhledem k tomu, že agenti jsou navrženi jako systémoví démoni, je možné, že budou běžet po celou dobu běhu systému. Neuvolňování paměti by v tomto případě mohlo zbytečně spotřebovávat celkovou operační paměť. Řešením by mohlo být automatické odstranění dlouho nepoužitých struktur z paměti, což je obtížně realizovatelné, protože Matahari agent nemá možnost zjistit, zda je možné paměť bezpečně uvolnit.

Stavovost funkcí Augeasu přináší možné bezpečnostní riziko, se kterým je nutno počítat. Komunikace přes D-Bus je nešifrovaná a veřejná pro kohokoli v systému. Útočník tedy může naslouchat volání metody `init` konfiguračního agenta a poté s vráceným identifikátorem relace provést vlastní změny v libovolném konfiguračním souboru. Právoplatný uživatel při uložení stromu konfiguračních souborů nevědomě uloží i změny provedené útočníkem. Řešení spočívající v nutnosti autorizovat provedení každé akce samostatně není možné, protože by se uživatel musel neustále autentizovat. Byla přijata dvě opatření proti tomuto útoku. První z nich spočívá v omezení rozsahu působení Augeasu tak, že je funkci `init` předán název jedné konkrétní lens definice, která je jediná načtená a pouze s ní lze pracovat. Druhé opatření spočívá ve svázání identifikace relace s názvem objektu volajícího programu, který je unikátní. Tento vztah je kontrolován při každém volání libovolné metody a pokud se liší název objektu volajícího a vlastníka, akce se neprovede a je vrácena chyba.

Kapitola 6

Testování

Nedílnou součástí každého počítačového systému by měl být proces testování. Obvykle se uvádí, že testování by mělo zaujímat padesát procent celkového času práce na programu [31]. Testování se skládá z několika částí, z nichž některé se nemusí uplatnit vždy. Nejdůležitějším hlediskem je, zda daný systém splňuje zadané požadavky, tedy jsou-li implementovány všechny požadované funkce. Dále je nutné ověřit, zda jsou jednotlivé funkce systému opravdu pracují správně a to i pro nevalidní vstupy a neočekávané stavy systému. Další kritéria systému, na která se proces testování obvykle také zaměřuje, mohou být stabilita, výkonnost, interoperabilita a další.

Testování prostupuje všemi částmi procesu tvorby programu, přičemž cena za opravu chyby v průběhu vývoje roste [9]. Opravení chyby v době návrhu je většinou bezbolestné, avšak po nasazení programu koncovým uživatelům je takováto oprava často velmi časově (tedy i finančně) náročná. Obzvláště pokud se jedná o chybu v návrhu aplikace. Opravit takovouto chybu není v některých případech ani možné, protože by znamenala přepsání značné části programu.

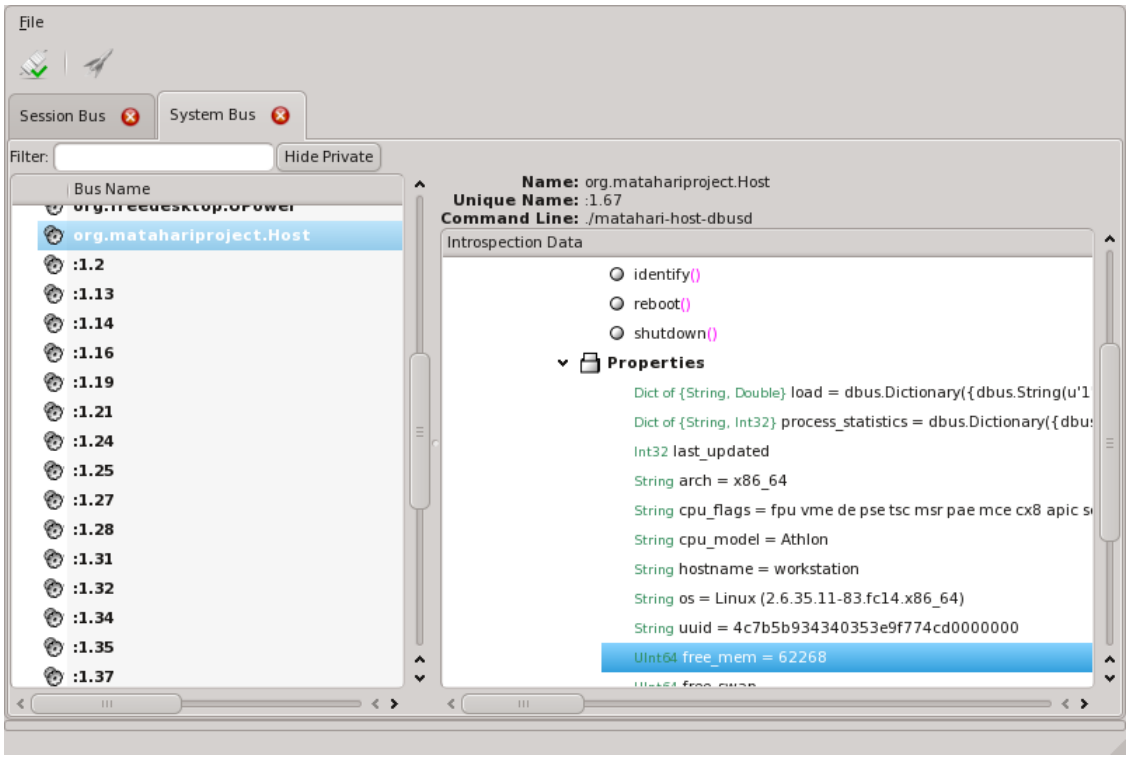
V první polovině této kapitoly je popsáno manuální testování, pomocí kterého byly průběžně testovány jednotlivé funkce programu při implementaci. Dále se tato kapitola zabývá testováním automatickým, které má zaručit, že se již implementované funkce nepřestanou po změnách v programu fungovat.

6.1 Manuální testování

Systém Matahari byl testován v několika krocích. V počátečních fázích implementace bylo prováděno převážně ruční testování, kdy se ověřovalo, zda funkce jednotlivých agentů provádí zadaný úkol a vrací správné hodnoty. Například zda agent pro obsluhu systémových služeb při zavolání funkce pro vypnutí určité služby ji opravdu vypne a vrátí správnou návratovou hodnotu. Knihovny implementující funkce jednotlivých agentů byly testovány tak, že se nad nimi napsal program, který výstupy jednotlivých metod a hodnoty vlastností objektů vypisoval na standardní výstup. Tento výstup byl poté manuálně porovnán s očekávanými výsledky.

Po vytvoření rozhraní se systémy D-Bus a QMF byly k testování použity nástroje pracující s těmito systémy. Pro D-Bus byl použit program `dbus-send`, který je součástí distribuce sběrnice D-Bus a slouží k volání rozhraní systému D-Bus z příkazové řádky. Uživatelsky přívětivější je použití nástroje s grafickým uživatelským rozhraním, který pomocí introspekce umožňuje zobrazit všechny dostupné metody a vlastnosti objektů daného rozhraní. Tako-

výchto nástrojů existuje celá řada, například `d-feet` nebo `qdbusviewer`. Na obrázku 6.1 je zobrazeno testování modulu *Host* v programu `d-feet`. Na levé straně okna je seznam všech objektů na sběrnici D-Bus. Po vybrání objektu se v pravé části zobrazí seznam metod a vlastností daného objektu. Po vybrání určité vlastnosti se zobrazí její hodnota. Při volání funkce se do okna zadají parametry, s jakými se funkce má volat. Pro systém QMF je možné použít nástroj `qmf-tool`, který umožňuje přistupovat ke QMF objektům pomocí parametrů spuštění nebo použitím vestavěného shellu.

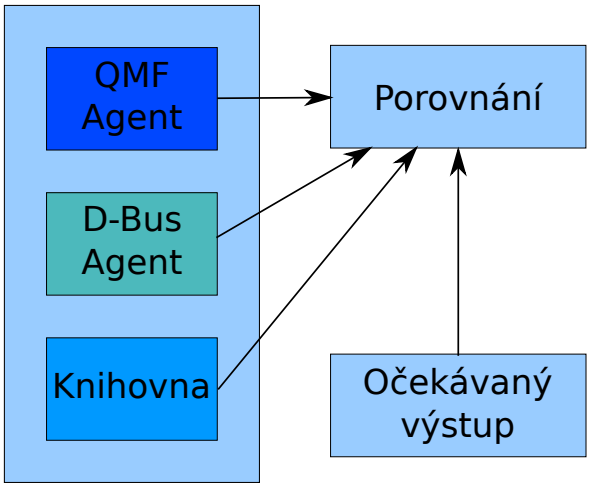


Obrázek 6.1: Testování D-Bus rozhraní v programu D-Feet.

6.2 Automatické testy

Automatické testy byly přidány v pozdější části vývoje, kdy nastala potřeba regresního testování. Regresní testování spočívá v opakovaném spouštění testů kvůli zajištění, že žádná provedená změna neovlivňuje funkci programu jinak, než bylo zamýšleno [28]. Pro automatizaci testování systému Matahari byla použita kombinace skriptu v jazyce Python a shellových skriptů. Shellové skripty slouží ke spuštění Qpid serveru na zadaném portu, pokud neběží. Dále tento skript spustí QMF agenty, D-Bus agenti se spustí automaticky ve chvíli, kdy je zavolána jakákoliv jejich metoda nebo je přistoupeno k vlastnosti objektu. Dalším krokem je spuštění testu v jazyce Python, který ověří, zda mají QMF i D-Bus agenti stejné výstupy a porovná je i s výstupem knihovny, viz obrázek 6.2. Porovnání ověří, zda se výsledky z jednotlivých částí shodují. Když se neshodují, jsou vypsány na standardní výstup a musí být vhodně interpretovány. Například funkce pro zjištění volné paměti vrací často různý výsledek pro volání jednotlivých částí, protože není volána ve stejném okamžiku a mezi tím může dojít ke změně. Nejedná se tedy o chybu. Pokud je to možné, zjistí tento

skript i očekávaný výsledek testování jiným způsobem, než je implementován v knihovně. Toto opatření slouží pro kontrolu, zda jsou funkce správně implementovány a nemění se jejich chování s různými verzemi podpůrných knihoven.



Obrázek 6.2: Schéma fungování automatického testování

K testování výstupů knihovny je použit modul *ctypes* ze standardní knihovny jazyka Python. Tento modul umožňuje volat funkce ze sdílených knihoven dynamicky bez nutnosti linkování. Je však nutné specifikovat typy argumentů a návratových hodnot volaných funkcí. Pro převod datových typů jsou v knihovně ctypes k dispozici ekvivalentní objekty k běžným typům jazyka C. Pro strukturované datové typy je možné použít třídu `Structure`, jejímž zděděním a definicí vlastností objektu lze vytvořit typ odpovídající libovolné struktuře jazyka C. Tato knihovna také podporuje ukazatele a tím i předávání parametrů odkazem. Kombinací těchto konstrukcí je možné pokrýt většinu požadavků pro testování knihoven projektu Matahari.

Nevýhoda tohoto skriptu je ta, že není integrován s žádným nástrojem pro automatické testování. Není jej proto možné snadno spouštět distribuovaně v rámci testování celého systému. Také je nutné výsledky testování ověřovat samostatně. Proto vznikají další testy, které využívají knihovnu *BeakerLib*. Jedná se sadu podpůrných skriptů v shellu pro usnadnění vytváření, spouštění testů a analýzu jejich výsledků [60]. Tato knihovna se skládá ze čtyř částí. *Journal* obsahuje podporu pro ukládání průběhu a výsledků testů do XML pro další použití. *Phases* definuje jednotlivé fáze testování, což umožňuje návrat do výchozího stavu, pokud určitá část testování selže. Pro snadnou kontrolu existence a obsahu souborů, návratových kódu programů atd., je možné použít modul *Asserts*. Poslední z částí knihovny BeakerLib je *Helpers*, jenž obsahuje pomocné funkce například pro zálohování a obnovu souborů, startovaní a ukončovaní služeb a další.

6.3 Výsledky

Skript v jazyce Python je určen pro kontrolu, zda je systém Matahari správně nainstalován a funguje. Může být spouštěn i na produkčních strojích, pokrývá proto kontrolu hodnot všech vlastností, ale testuje pouze výstupní parametry některých funkcí. Jsou volány pouze funkce, které neovlivňují běh testovacího počítače. Například tyto testy nepokrývají testování funkcí pro zapnutí a vypnutí systémových služeb, protože nelze určit, které služby

je možné vypnout bez následků. Například vypnutí SSH démona na serveru bez fyzického přístupu by znemožnilo přihlášení. Je však v plánu testy pro tyto funkce přidat to testovacích skriptů založených na knihovně BeakerLib. Tyto testy pak budou spouštěny na dedikovaných testovacích strojích, kde selhání některých funkcí neovlivní funkci produkčního stroje.

Kapitola 7

Rozšíření

Systém Matahari byl navržen tak, aby byl co nejsnadněji rozšířitelný. Je možné na Matahari pohlížet jako na nástroj pro podporu vytváření konfiguračních agentů. Pokud kdokoli chce vytvořit Matahari agenta má dvě možnosti. Buď může přidat agenta k existujícím agentům, které jsou součástí projektu Matahari, nebo nechat agenta samostatně a pouze využít prostředky, které Matahari poskytuje. První možnost je výhodná v tom, že po začlenění do projektu Matahari odpadá nutnost vlastní distribuce a je zaručeno, že bude agent přizpůsoben změnám v projektu Matahari. Druhý přístup je flexibilnější v tom, že agent může poskytovat funkce přesně na míru projektu, který jej bude využívat. Pro začlenění do Matahari je vhodné, aby bylo rozhraní univerzálnější. Nicméně není problém přejít z druhého řešení k prvnímu, například po odladění funkcí agenta. Opačný směr však možný není kvůli nutnosti zachování kompatibility.

Další vývoj projektu Matahari se bude odebírat dvěma směry. Jednak se budou přidávat další agenti, kteří dále rozšíří poskytované rozhraní, jednak se bude přidávat podpora pro další operační systémy a jejich verze. V současné době je projekt Matahari odladěn na systémech Fedora, Red Hat Enterprise Linux a Microsoft Windows XP, v budoucnu se počet podporovaných systému bude dále rozšiřovat.

V této kapitole bude nejprve nastíněno, jak vytvořit samostatného agenta, který bude využívat prostředky systému Matahari. Dále budou diskutovány možnosti rozšíření, tedy které agenty by bylo vhodné implementovat a k čemu by sloužili.

7.1 Vytvoření samostatného Matahari agenta

Při vytváření samostatného Matahari agenta je možné pro usnadnění využít nástrojů, které Matahari poskytuje. Při použití překladového systému CMake je k dispozici několik maker, která zautomatizují některé části překladu. Makro `generate_qmf_schemas` nejprve zjistí, zda byl od posledního použití změněn definiční XML soubor pro QMF. Pokud ano, přegeneruje pomocí nástroje `qmf-gen` zdrojové soubory, které obsahují definici třídy *PackageDefinition*. Tato třída obsahuje datové struktury, které odpovídají danému schématu. Ty jsou poté využity při implementaci tohoto rozhraní. Pro D-Bus rozhraní jsou k dispozici makro `generate_dbus_headers`, které XML definice D-Bus rozhraní vytvoří hlavičkové soubory. Ty jsou poté využity pro automatickou transformaci datových typů sběrnice D-Bus a jazyka C a řízení, která metoda se má při zavolat při určitém volání přes D-Bus. Dále vygenerovaný kód obsahuje obsluhu řízení

7.2 Agent pro přístup k logovacím souborům

Vzhledem k tomu, že projekt Matahari je mimo jiné zaměřen na podporu nástrojů pro správu velkého množství jak virtuálních, tak i fyzických strojů, je podpora pro přístup logům důležitá. Správce těchto systému si může po sesbírání těchto dat na jednom místě zobrazit všechny chybové hlášky a snadno odhalit potenciální problémy dříve, než se projeví. Například si může zobrazit pokusy o nepovolený přístup z logu SSH démona a odhalit pokus o kompromitaci systémů.

Další rozšíření tohoto agenta, které je v současné době ve fázi plánování, je přidání podpory pro systém ABRT. ABRT je zkratka z Automatic Bug Reporting Tool, čili automatický nástroj pro reportování chyb. Tento program zachytává nestandardní ukončení programů a vytváří zprávy o chybě, aby vývojáři mohli chybu zreprodukovat a opravit. Integrace s agentem pro přístup logovacím souborům by spočívala v tom, že mimo logů by tento agent vracel i chybová hlášení sesbíraná programem ABRT. Tato hlášení mohou odhalit jinak těžko odhalitelné chyby. V některých případech chyby segmentace při špatnému přístupu do paměti mohou být zneužitelné. Útočník může těchto chyb využít ke spuštění vlastního kódu, což je vážný bezpečnostní problém. Pokud však bude mít správce systému dostatečné informace o chybě, může ji opravit dříve, než k útoku dojde.

7.3 Agent pro správu software

Mezi další časté úkony správců systémů patří správa software. Je třeba udržovat software v aktuálních verzích, instalovat bezpečnostní opravy a zjišťovat informace o verzích programů na jednotlivých systémech. Správce si například může zobrazit, které systémy nemají opravenou verzi určitého programu a mohou být zranitelné. Problém však nastává s požadavkem na podporu co největšího množství platforem. Vytvořit agenta, který bude umět získávat informace o balíčcích a instalovat je v různých linuxových distribucích je sice komplikované, nicméně ne neřešitelné. Problém je s podporou systému Windows, které nemají možnost správy softwarových balíčků. Je možné spolupracovat s Windows Update na opravách operačního systému, ale spravovat jednotlivé verze programů je problém, protože neexistuje podporovaná alternativa repozitářů softwaru známých z běžných linuxových distribucí. Z tohoto důvodu bude nejprve tento agent vytvořen pouze pro linuxové systémy. Mezi jednotlivými distribucemi jsou však výrazné rozdíly ve správě balíčků, takže vytvoření univerzálního agenta pro všechny distribuce není možné. Je však také nutné zvolit, na jaké úrovni bude agent pracovat. Pokud by pracovat pouze na úrovni jednotlivých balíků, stačila by podpora pro balíky RPM a DEB a pokryl by tím většinu běžně používaných distribucí. Vhodnější by však bylo pracovat na úrovni repozitářů, což by umožnilo především využít kanály pro distribuci balíčků nabízené tvůrci distribuce, čímž se zjednoduší práce správcům systému. Avšak toto řešení je náročnější, protože je nutné vytvořit podporu pro různé nástroje pro správu balíků, které se výrazně liší mezi distribucemi. Možným řešením by bylo využití projektu PackageKit [14], což je abstrakce nad správci balíčků. PackageKit poskytuje jednotné rozhraní pro základní operace pro manipulaci se balíčky softwaru a zapouzdřuje rozdíly mezi jednotlivými správci. Jeho využití by výrazně usnadnilo vytvoření agenta pro správu software.

7.4 Nástroj pro správu systémů

Konečným cílem, ke kterému Matahari poskytuje potřebné nástroje, je vytvoření nástroje pro správu a monitorování systémů. Tento nástroj bude sbírat data od jednotlivých agentů a podle zadaných požadavků jim určovat, co mají dělat. Pomocí jednotného rozhraní bude možné spravovat jak unixové systémy, tak i systémy Microsoft Windows. V dnešní době je trend vytvářet pro tyto nástroje webové rozhraní, které umožní přehledně zobrazovat nejen stav jednotlivých stanic, ale i přehledy pro všechny systémy, případně jejich skupiny. Další možnost je upravit stávající nástroje pro správu tak, aby spolupracovali s Matahari. Jedním ze současných cílů projektu FMCI je upravit některé současné nástroje pro správu lokálních systémů, tak aby využily Matahari agenty. Podle potřeby budou vytvořeny i některé nové nástroje pro usnadnění nastavení počítače i méně zkušeným uživatelům.

Kapitola 8

Závěr

Tato práce si klade za cíl rozšířit existující nástroje pro správu o nový nástroj, který bude použitelný pro správu jak lokální, tak i vzdálenou se zaměřením na cloud computing. Tento nástroj poskytne systémové démony pro základní úlohy konfigurace a monitorování systémů a platformu pro vytváření dalších démonů pro správu.

Problematika systémových rozhraní a komunikace je diskutována v kapitole 2. Jsou zde také uvedeny některé nástroje pro správu a způsob jejich komunikace. Základní terminologie cloud computingu je popsána v kapitole 3.

Popis architektury navrženého systému je obsahem kapitoly 4. Nejdůležitějším rysem návrhu je modulárnost systému a možnost přizpůsobení pro potřeby různých projektů. Implementace je popsána v kapitole 5. Implementované části jsou dostupné na přiloženém CD a v repozitářích na serveru GitHub.

Testování probíhalo v několika krocích. V počáteční fázi vývoje se převážně testovalo ručně, později přibyla nutnost regresního testování. Proto byl vytvořen program, který testuje jednotlivé části systému a porovnává výsledky mezi nimi. Testování je podrobně popsáno v kapitole 6.

V kapitole 7 je naznačeno další směřovaní projektu a navrženo několik dalších rozšíření.

Všechny body zadání byly splněny. Vytvořený systém je snadno použitelný pro základní konfiguraci lokálních i vzdálených systémů. Následujícím krokem je vytvoření nástrojů s grafickým uživatelským rozhraním pro zpřístupnění tohoto systému širší skupině uživatelů, případně úprava stávajících nástrojů tak, aby tento systém využily. Další vývoj bude také probíhat rozšířením možností konfigurace systému. Budou přidáni další systémoví démoni pro správu různých častí systému.

Literatura

[1] BENSON, C.; ELMAN, A.; NICKELL, S.; aj.: *GNOME Human Interface Guidelines*. 2010, [online. citováno dne 16.3.2011]. http://library.gnome.org/devel/hig-book/stable/

[2] BRINDLEY, L.; ROBIE, J.: *Red Hat Enterprise MRG 1.3: Messaging User Guide*. Red Hat, 2010.

[3] BURTON, R.: Connect desktop apps using D-BUS. *IBM developerWorks*, 2004, [online. citováno dne 19.10.2010]. http://www.ibm.com/developerworks/linux/library/l-dbus.html

[4] CAMERON, J.: *Managing Linux systems with Webmin*. Pearson Education, Inc., 2004, ISBN 0-13-140882-8.

[5] CHANG, W.; ABU-AMARA, H.; SANFORD, J.: *Transforming Enterprise Cloud Services*. Springer, 2010, ISBN 978-90-481-9845-0.

[6] CHAPPELL, D.: Introducing the Windows Azure Platform. Technická zpráva, Microsoft Corporation, 2010.

[7] CHE, B.: *Red Hat Enterprise MRG: Messaging, Realtime, and Grid*. Red Hat, 2009.

[8] COOPER, J.: *The book of Webmin*. Linux Journal Press, 2003, ISBN 1-886411-92-1.

[9] DUSTIN, E.: *Effective software testing: 50 specific ways to improve your testing*. Pearson Education, Inc., 2003, ISBN 0-201-79429-2.

[10] FOSTER, I.; ZHAO, Y.; RAICU, I.; aj.: *Cloud Computing and Grid Computing 360-Degree Compared*. IEEE Grid Computing Environments Workshop, 2008.

[11] FOSTER, N.; PIERCE, B.; GREENWALD, M.; aj.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 2005, ISSN 0164-0925.

[12] FURHT, B.; ESCALANTE, A.: *Handbook of Cloud Computing*. Springer, 2010, ISBN 978-1-4419-6524-0.

[13] GILLAM, L.: *Cloud Computing: Principles, Systems and Applications*. Springer, 2010, ISBN 978-1-84996-240-7.

[14] HARNAL, A. K.: *Linux Applications And Administration*. Tata McGraw-Hill, 2009, ISBN 978-0-07-065675-8.

[15] HUANG, Y. M.; NIE, Z. H.: Cloud computing with Linux and Apache Hadoop. 2009, [online. citováno dne 12.12.2010].
https://www.ibm.com/developerworks/aix/library/au-cloud_apache/?ca=dth-cloud&S_TACT=105AGX30&S_CMP=DevX4Q

[16] HURWITZ, J.; BLOOR, R.; KAUFMAN, M.; aj.: *Cloud Computing For Dummies*. Wiley Publishing, Inc., 2010, ISBN 978-0-470-48470-8.

[17] ISO/IEC: *ISO/IEC TR 10182. Information Technology – Programming languages, their environments and system software interfaces - Guidelines for language bindings*. ISO/IEC, 1993.

[18] JANG, M.: *Ubuntu Server Administration*. McGraw-Hill Osborne Media, 2009, ISBN 978-0-07-159892-7.

[19] JENNINGS, N. R.; WOOLDRIDGE, M. J.: *Agent technology: foundations, applications, and markets*. Springer, 2002, ISBN 3-540-63591-2.

[20] KORPELA, E.; WERTHIMER, D.; ANDERSON, D.; aj.: SETI@home — Massively Distributed Computing for SETI. *Scientific Programming*, January/February 2001.

[21] KRAMER, J.: Advanced Message Queuing Protocol. *Linux Journal*, November 2009, ISSN 1075-3583.

[22] KREJČÍ, R.: D-Bus s využitím knihovny libdbus. 2008, [online. citováno dne 25.10.2010].
http://www.radekkrejci.net/cz/knowhow/dbus

[23] LAIRD, P.: Cloud Computing Taxonomy. 2009, [online. citováno dne 12.12.2010].
http://peterlaird.blogspot.com/2009/05/cloud-computing-taxonomy-at-interop-las.html

[24] LEON, E. D.: The Five Layers within Cloud Computing. 2009, [online. citováno dne 12.12.2010].
http://cloudcomputing.sys-con.com/node/1200642

[25] LIPOVSKÝ, J.: *Jednoduché sdílení pomocí SMB protokolu ve správci Nautilus*. Bakalářská práce, FIT VUT v Brně, Brno, 2010.

[26] LOVE, R.: Get on the D-BUS. *Linux Journal*, February 2005, ISSN 1075-3583.

[27] LUTTERKORT, D.: *Augeas – a configuration API*. Proceedings of the Linux Symposium, 2008.

[28] MALHOTRA, J. J.; TIPLE, B. S.: *Software Testing and Quality Assurance*. Nirali Prakashan, 2008, ISBN 978-81-906396-4-4.

[29] MENKEN, I.; BLOKDIJK, G.: *Virtualization: The Complete Cornerstone Guide to Virtualization Best*. Emereo Pty Ltd, 2008, ISBN 1921523913.

[30] van der MOLEN, F.: *Get Ready for Cloud Computing*. Van Haren Publishing, 2010, ISBN 978-90-8753-640-4.

[31] MYERS, G. J.; BADGETT, T.; THOMAS, T. M.; aj.: *The art of software testing*. John Wiley & sons, Inc., 2004, ISBN 0-471-46912-2.

[32] MYERS, P.: *Matahari: Remote APIs for Systems Management*. Red Hat, 2011.

[33] NEGUS, C.: *Linux Bible 2011 Edition*. Wiley Publishing, Inc., 2011, ISBN 978-0-470-92998-8.

[34] NEMETH, E.; SNYDER, G.; HEIN, T.: *Linux administration handbook*. Pearson Education, Inc., 2007, ISBN 0-13-148004-9.

[35] PALMIERI, J.: Get on D-BUS. *Red Hat Magazine*, January 2005.

[36] PENNINGTON, H.; WHEELER, D.; PALMIERI, J.; aj.: D-Bus Tutorial. 2011, [online. citováno dne 18.3.2011].
http://dbus.freedesktop.org/doc/dbus-tutorial.html

[37] PERCUMA, I.: Cloud Computing. 2010, [online. citováno dne 12.12.2010].
http://infolayan.blogspot.com/2010/05/cloud-computing.html

[38] PETERSEN, R.: *Fedora 11: Administration, Networking, Security*. Surfing Turtle Press, 2009, ISBN 978-0-9820998-8-9.

[39] RICHARDSON, L.; RUBY, S.: *RESTful web services*. O'Reilly Media, Inc., 2007, ISBN 978-0-596-52926-0.

[40] SICAM, C.; BACLIT, R.; MEMBREY, P.; aj.: *Foundations of CentOS Linux: Enterprise Linux On the Cheap*. Apress, Inc., 2009, ISBN 978-1-4302-1964-4.

[41] TERPSTRA, J.; VERNOOIJ, J.: *The official Samba-3 HOWTO and reference guide*. Prentice Hall PTR, 2004, ISBN 0-13-145355-6.

[42] The Apache Software Foundation: *AMQP Messaging Broker (Implemented in C++)*. 2010.

[43] WELTE, A.; WELTE, T.; ELSENPETER, R.: *Cloud Computing: A Practical Approach*. McGraw Hill Professional, 2009, ISBN 978-0-07-162694-1.

[44] WHEELER, D.: DTD for D-Bus Introspection data. 2005, [online. citováno dne 6.1.2011].
http://standards.freedesktop.org/dbus/1.0/introspect.dtd

[45] WILLIAMS, D.: *Virtualization with Xen: including XenEnterprise, XenServer, and XenExpress*. Syngress Publishing, Inc., 2007, ISBN 978-1-59749-167-9.

[46] WWW stránky: Apache Qpid: Open Source AMQP Messaging - Qpid Management Framework. 2008, [online. citováno dne 6.1.2011].
https://cwiki.apache.org/qpid/qpid-management-framework.html

[47] WWW stránky: Advanced Message Queuing Protocol. 2010, [online. citováno dne 11.11.2010].
http://www.amqp.org/confluence/display/AMQP

[48] WWW stránky: Amazon EC2 FAQs. 2010, [online. citováno dne 30.12.2010].
http://aws.amazon.com/ec2/faqs/#What_operating_system_environments_are_supported

[49] WWW stránky: Amazon Elastic Compute Cloud (Amazon EC2). 2010, [online. citováno dne 30.12.2010].
http://aws.amazon.com/ec2/

[50] WWW stránky: D-Bus Server Design Issues. 2010, [online. citováno dne 14.4.2011].
http://wiki.maemo.org/Documentation/Maemo_5_Developer_Guide/DBus/D-Bus_Server_Design_Issues

[51] WWW stránky: freedesktop.org - Software/dbus. 2010, [online. citováno dne 14.10.2010].
http://www.freedesktop.org/wiki/Software/dbus

[52] WWW stránky: Google App Engine — Google Code. 2010, [online. citováno dne 30.12.2010].
http://code.google.com/intl/cs/appengine/

[53] WWW stránky: Matahari Overview. 2010, [online, citováno dne 28.12.2010].
https://fedorahosted.org/matahari/#Overview

[54] WWW stránky: Private or Public Clouds? 2010, [online. citováno dne 10.5.2011].
http://nskinc.blogspot.com/2010/06/public-or-private-cloud.html

[55] WWW stránky: SIGAR - System Information Gatherer And Reporter. 2010, [online. citováno dne 19.2.2011].
http://support.hyperic.com/display/SIGAR/Home

[56] WWW stránky: spacewalk. 2010, [online. citováno dne 1.1.2011].
https://fedorahosted.org/spacewalk/

[57] WWW stránky: SystemConfigCleanup. 2010, [online, citováno dne 27.12.2010].
http://fedoraproject.org/wiki/Features/SystemConfigCleanup

[58] WWW stránky: Welcome to FMCI. 2010, [online, citováno dne 28.12.2010].
https://fedorahosted.org/fmci

[59] WWW stránky: Augeas. 2011, [online. citováno dne 31.1.2011].
http://augeas.net/index.html

[60] WWW stránky: BeakerLib. 2011, [online. citováno dne 13.5.2011].
https://fedorahosted.org/beakerlib/wiki/Manual

[61] WWW stránky: Boomerang: A bidirectional programming language for ad-hoc data. 2011, [online. citováno dne 13.2.2011].
http://alliance.seas.upenn.edu/~harmony/

[62] WWW stránky: Puppet Labs Documentation. 2011, [online. citováno dne 24.3.2011].
http://docs.puppetlabs.com/

[63] ZEUTHEN, D.: PolicyKit Reference Manual. 2010, [online. citováno dne 20.12.2010].
http://hal.freedesktop.org/docs/polkit/polkit.8.html