VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2016 Bc. Jan Král

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

VIRTUALIZACE OPERAČNÍCH SYSTÉMŮ

VIRTUALIZATION OF OPERATING SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Jan Král

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Ing. Dan Komosný, Ph.D.

BRNO 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**
Ústav telekomunikací

Student: Bc. Jan Král ***ID:*** 146870
Ročník: 2 ***Akademický rok:*** 2015/16

NÁZEV TÉMATU:

Virtualizace operačních systémů

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s principem virtualizace operačních systémů. Realizujte různé typy virtualizace operačního systému Linux, distribuce CentOS. Konfiguraci každého typu virtualizace upravte tak, aby bylo možno se vzdáleně připojit na virtualizovaný operační systém pomocí SSH (Secure Shell). Porovnejte parametry vzdáleného připojení (zejména časovou prodlevu pro připojení) na operační systém bez virtualizace a s různými typy virtualizace.

DOPORUČENÁ LITERATURA:

[1] PUŽMANOVÁ, R. TCP/IP v kostce. 2. vyd. Kopp, 2009. 620 s. ISBN: 978-80-7232-388-3.

[2] Linux Dokumentační projekt. 4. vyd. Computer Press, 2008. 1336 s. ISBN: 978-80-251-1525-1.

[3] COOPER, M. Advanced Bash-Scripting Guide. Steve Glines, 2010. 518 s. ISBN: 978-14-357-5218-4.

Termín zadání: 1.2.2016 ***Termín odevzdání:*** 25.5.2016

Vedoucí práce: doc. Ing. Dan Komosný, Ph.D.
Konzultant diplomové práce:

doc. Ing. Jiří Mišurec, CSc., *předseda oborové rady*

UPOZORNĚNÍ:
Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Diplomová práce „Virtualizace operačních systémů“ se v úvodu věnuje obecnému popisu virtualizačních technologií a možnostem jejich využití. V práci jsou také uvedeny příklady virtualizačních technologií různých typů, včetně detailnějšího popisu nástrojů KVM a Docker použitých při měření. Dále je v této práci navržen a realizován postup měření vlivu výše zmíněných virtualizačních nástrojů na síťové služby SSH, Telnet, FTP a odezvu síťového rozhraní v rámci virtuálních stanic. Pro automatizované měření požadovaných hodnot byla vytvořena aplikace, která je v této práci také detailně popsána. Práce dále obsahuje detailní rozbor, analýzu a prezentaci všech naměřených hodnot. Závěr práce shrnuje dosažené výsledky a potvrzuje vliv virtualizace na parametry síťových služeb, kdy aplikační kontejnery služby Docker, které jsou svými nízkými režijními náklady na provoz srovnatelné se systémem bez virtualizace, vykazovaly znatelně lepší výsledky než tradiční virtuální stanice realizované pomocí virtualizace KVM.

KLÍČOVÁ SLOVA

Virtualizace, SSH, Telnet, FTP, ICMP, latence, výkon, KVM, Docker, kontejnery

ABSTRACT

The diploma thesis 'Virtualization of Operating Systems' deals with a general description of virtualization technology and briefly discusses its use cases and advantages. The thesis also mentions examples of different types of virtualization technologies and tools, including more thorough description of the two technologies used for measurement: Docker and KVM. The second part of this thesis descibes the preparation, installation and configuration of all the tools and services that are necessary for measurement of the influence of the aforementioned virtualization technologies on network services running on the virtual machines, including analysis and discussion of the resulting data. Moreover, a custom application for fully automated measurement of the parameters of network services was created, and is also described in this thesis. The conclusion of this thesis summarizes and discusses the achieved results and acknowledges the influence of virtualization on network services, where the Docker application containers, which are with their low overhead comparable to a "bare" system without any virtualization, managed to achieve much better performance results than the traditional virtual machines on KVM.

KEYWORDS

Virtualization, SSH, Telnet, FTP, ICMP latency, performance, KVM, Docker, containers

KRÁL, Jan *Virtualizace operačních systémů*: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2016. 97 s. Vedoucí práce byl doc. Ing. Dan Komosný, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Virtualizace operačních systémů“ jsem vypracoval(a) samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor(ka) uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil(a) autorská práva třetích osob, zejména jsem nezasáhl(a) nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom(a) následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora(-ky)

PODĚKOVÁNÍ

Toto poděkování bych chtěl věnovat vedoucímu této diplomové práce, panu doc. Ing. Danu Komosnému, Ph.D. za jeho vedení a cenné a podnětné návrhy a rady k této práci.

Brno . . . .

podpis autora(-ky)

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic
http://www.six.feec.vutbr.cz

PODĚKOVÁNÍ

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno ....
podpis autora(-ky)

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OP Výzkum a vývoj
pro inovace

OBSAH

Úvod **12**

1 Síťové služby a virtualizace **13**

1.1 Typy virtualizace . . . 13
1.1.1 Hardwarová a softwarová virtualizace . . . 13
1.1.2 Virtualizace pracovních stanic . . . 15
1.1.3 Virtualizace aplikací . . . 15
1.1.4 Ostatní typy virtualizace . . . 16
1.1.5 Virtualizační metody použité při měření . . . 17
1.2 Protokoly a služby použité při měření . . . 20
1.2.1 Protokol ICMP . . . 20
1.2.2 Protokol FTP . . . 22
1.2.3 Služba SSH . . . 24
1.2.4 Služba Telnet . . . 25

2 Realizace měřícího pracoviště **27**

2.1 Příprava prostředí KVM/QEMU . . . 27
2.1.1 Instalace balíčků . . . 27
2.1.2 Konfigurace sítě . . . 28
2.1.3 Příprava výchozího obrazu disku – instalace systému . . . 29
2.1.4 Konfigurace služeb . . . 32
2.1.5 Finální úpravy obrazu disku . . . 34
2.1.6 Instalace virtuálních stanic . . . 35
2.2 Příprava prostředí Docker . . . 37
2.2.1 Instalace balíčků . . . 37
2.2.2 Konfigurace . . . 38
2.2.3 Příprava image . . . 39
2.2.4 Vytvoření image . . . 41
2.2.5 Instalace kontejnerů . . . 43
2.3 Vytvořená aplikace `vte` . . . 44
2.3.1 Popis modulů aplikace `vte` . . . 46
2.3.2 Výstupní data a popis naměřených hodnot . . . 62

3 Výsledky měření **66**

3.1 Srovnání nástrojů Docker a KVM . . . 67
3.1.1 Zátěž systému . . . 67
3.1.2 Ping . . . 70

3.1.3 FTP 70
3.1.4 SSH 74
3.1.5 Telnet 75
3.1.6 Zhodnocení 76
3.2 Výkon nástrojů Docker a KVM/QEMU při zatížení systému 78
3.2.1 Výkon při zatížení CPU 78
3.2.2 Výkon při zatížení operační paměti 79
3.2.3 Výkon při zatížení IO 80

4 Závěr 86

Literatura 87

Seznam symbolů, veličin a zkratek 89

Seznam příloh 92

A Obsah přiloženého DVD 93

B Aplikace vte 94
B.1 Instalace aplikace vte 94
B.2 Externí závislosti aplikace vte 95
B.2.1 Redis 95
B.2.2 Celery 96
B.3 Seznam hodnot výchozí konfigurace aplikace `vte` 96

SEZNAM UKÁZEK

1.1 Otestování spojení s HTTP serverem pomocí nástroje Telnet 26
2.1 Instalace virtualizačních balíčků 27
2.2 Skript `generate_network_config.py` pro automatické generování konfigurace sítě 29
2.3 Vygenerovaná konfigurace sítě 30
2.4 Použití nástroje `virt-install` pro vytvoření VM 30
2.5 Příprava pro přihlašování k SSH serveru pomocí klíče 32
2.6 Konfigurace telnet serveru 32
2.7 Konfigurace FTP serveru 33
2.8 Část konfigurace nástroje `supervisord` 34
2.9 Komprese obrazu disku pomocí nástroje `qemu-img` 34
2.10 Použití nástroje `virt-sysprep` 35
2.11 Skript `set_hostname.sh` 35
2.12 Použití nástroje `virt-sysprep` pro jednorázové spuštění skriptu. . . . 35
2.13 Skript `vm_install.py` pro automatické vytvoření zadaného počtu VM 36
2.14 Instalace balíčků Docker 37
2.15 Ověření funkčnosti nástroje Docker pomocí předpřipraveného kontejneru 38
2.16 Konfigurace parametrů Docker démona 39
2.17 První část souboru Dockerfile 39
2.18 Druhá část souboru Dockerfile 40
2.19 Závěrečná část souboru Dockerfile 41
2.20 Proces vytvoření image pomocí příkazu `docker build` 42
2.21 Seznam dostupných image 42
2.22 Seznam vrstev image `kral/centos-all` 43
2.23 Vytvoření kontejnerů z image `kral/centos-all` 44
2.24 Nápověda nástroje `vte` 45
2.25 Kód dekorátoru `timed` 51
3.1 Část výstupu nástroje `iotop` během měření 82
B.1 Spuštění aplikace Redis 95
B.2 Spuštění procesu `celery worker` 96
B.3 Výchozí konfigurace aplikace `vte` 97

SEZNAM OBRÁZKŮ

1.1 Rozdělení hypervizorů na typy 14
1.2 Rozdíl mezi HW/SW virtualizací a virtualizací aplikací 16
1.3 Schéma virtualizace KVM 18
1.4 Architektura nástroje Docker 19
1.5 Záhlaví ICMP paketu 21
1.6 Režimy přenosu FTP souborů 23
1.7 Zachycená data protokolu Telnet 26
2.1 Rozdělení aplikace `vte` do modulů 46
3.1 Graf využití CPU při jednotlivých měřeních 68
3.2 Graf zátěže systému v intervalu 15 min při jednotlivých měřeních . . 68
3.3 Graf využití operační paměti při jednotlivých měřeních 69
3.4 Síťová odezva VM 70
3.5 Doba stahování souboru z FTP serveru pro jednotlivé virtualizace . . 71
3.6 Doba nahrání souboru na FTP server 72
3.7 Prodleva na straně klienta při nahrávání souboru na FTP server . . . 74
3.8 Graf naměřených hodnot pro službu SSH 74
3.9 Naměřené hodnoty pro službu Telnet 77
3.10 KVM – Vliv zatížení CPU na dobu stahování souboru z FTP serveru 79
3.11 KVM – Vliv zatížení CPU na dobu trvání SSH spojení 80
3.12 Docker – Doba nahrávání souboru na FTP server podle vytížení CPU 81
3.13 Docker – Doba trvání SSH spojení podle vytížení CPU 82
3.14 KVM – Zátěž systému v závislosti na počtu aktivních VM a využité paměti RAM 83
3.15 KVM – Procento využité paměti RAM 83
3.16 KVM – Doba spojení SSH v závislosti na využité paměti a počtu aktivních VM 84
3.17 Docker – doba trvání SSH spojení v závislosti na využití paměti a počtu aktivních VM 84
3.18 KVM – Závislost procesorového času stráveného ve stavu `iowait` na počtu VM a zatížení disků 85
3.19 KVM – Závislost doby stahování souboru z FTP serveru 85

SEZNAM TABULEK

3.1 Parametry aplikace `stress` použité při měření 66
3.2 Analýza měření protokolu FTP 72

ÚVOD

Virtualizace je technologie, která má v současnosti velmi široký záběr a velký počet možností využití. Od virtualizace serverů, přes koncové stanice a úložiště, až po virtualizaci sítí a aplikací, nebo dokonce celé výpočetní infrastruktury, se její popularita a rozšíření neustále rozrůstá, ať už díky úsporám nákladů na fyzickou infrastrukturu, zajištění bezpečnosti, nebo simulaci hardwaru.

Tato práce je zaměřena zejména na chování této technologie v reálném provozu. Jejím cílem je zjistit a porovnat, jaký vliv má virtualizace na parametry různých síťových služeb, např. vzdáleného připojení pomocí protokolů SSH a Telnet, nahrávání a stahování souborů ze vzdálené stanice pomocí protokolu FTP nebo na síťovou odezvu virtualizovaných stanic s operačním systémem Linux, distribucí CentOS.

První část této práce se věnuje stručnému popisu virtualizace a jejímu základnímu členění. Kromě klasických způsobů, zejména hardwarové a softwarové virtualizace, je zde zmíněna také virtualizace aplikací pomocí kontejnerů, která je v současnosti velmi populární, hlavně kvůli nástroji Docker, a dále je okrajově pojednáno i o některých dalších typech virtualizace. Virtualizační metody KVM a Docker, kterým se práce věnuje, jsou potom rozepsány podrobněji. První kapitola také obsahuje stručný popis všech síťových služeb, jejichž parametry byly v dalších částech práce srovnávány na různých typech virtualizace.

Druhá kapitola práce se věnuje realizaci měřícího pracoviště a samotnému způsobu měření. Je v ní popsána příprava prostředí pro zprovoznění virtualizačních nástrojů Docker a KVM, od instalace systémových balíčků, přes přípravu výchozích obrazů virtualizovaných stanic, až po jejich samotné zprovoznění. Tato kapitola také obsahuje podrobný popis aplikace, která byla v rámci této diplomové práce vytvořena za účelem automatického měření parametrů síťových služeb na virtualizovaných stanicích. Na závěr kapitoly je uveden detailní popis všech naměřených hodnot a metrik.

Třetí a poslední kapitola se potom věnuje analýze získaných a naměřených hodnot. Obsahuje srovnání parametrů síťových služeb provozovaných na virtuálních stanicích při použití virtualizace KVM i Docker pro různé úrovně zatížení hostitelského systému, a dále také krátkou analýzu vlivu extrémního zatížení různých částí systému (CPU, paměti a disků) na jednotlivé virtualizační technologie. Analýzou naměřených hodnot byl poté prokázán vliv virtualizace na parametry síťových služeb, kdy aplikační kontejnery Docker, které jsou se svými nízkými režijními náklady prakticky shodné se systémem bez použití virtualizace, dosahovaly výrazně lepších hodnot, než tradiční VM realizované pomocí virtualizace KVM.

1 SÍŤOVÉ SLUŽBY A VIRTUALIZACE

Koncept virtuálních stanic existuje již od roku 1960, kdy jej poprvé použila společnost IBM pro současný přístup více uživatelů k sálovým počítačům. Z pohledu uživatelů přistupovali přímo k fyzické stanici, ale ve skutečnosti jejich interakce probíhala s virtualizovanou instancí této stanice. Tehdy se jednalo o efektivní a jednoduchý způsob, jak umožnit sdílení zdrojů na nákladném hardwaru. Během 70. a 80. let minulého století virtualizace s příchodem levnějšího hardware ustoupila do pozadí, nicméně se následně s rozvojem různých architektur hardware a variant operačních systémů během let 90. opět vynořila na povrch – umožňovala provozovat širokou škálu aplikací cílených na jiný hardware a OS na dané stanici.

Virtualizace by se obecně dala popsat jako technologie, která umožňuje spojit nebo naopak rozdělit hardwarové zdroje (ať už výpočetní kapacitu, diskový prostor, nebo jiné) a následně je dát k dispozici různým prostředím pomocí členění hardwaru a softwaru, částečné nebo úplné simulace, emulace, nebo sdíleného přístupu (např. v čase), a dalších. Virtualizace má například následující uplatnění:

- sjednocení serverů – virtualizace umožňuje konsolidovat služby napříč větším počtem serverů, které nejsou plně využity, na menší počet, což šetří náklady a zdroje,
- vytváření izolovaných prostředí (sandboxing) – pomocí virtualizace je možné vytvořit plně izolované a zabezpečené výpočetní prostředí, které lze využít např. ke spouštění neověřených nebo nebezpečných aplikací a software,
- virtualizace hardware – pomocí virtualizace je možné vytvořit hardware, který by jinak nebyl k dispozici (např. virtuální síťovou infrastrukturu),
- souběh více operačních systémů – virtualizace umožňuje souběh více operačních systémů na jedné stanici,
- samostatné aplikace – virtualizace umožňuje vytváření „balíků“ aplikací společně s programovým prostředím nutným pro jejich běh,
- testovací prostředí – pomocí virtualizace je možné vytvářet různá testovací prostředí, která by bylo obtížné realizovat jiným způsobem.[1]

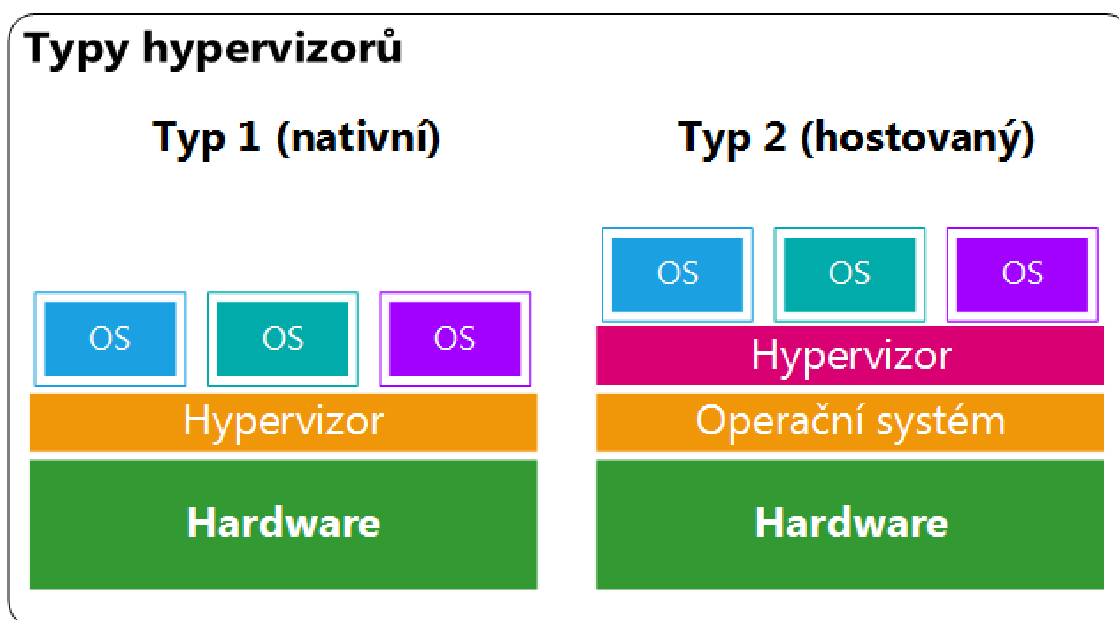
1.1 Typy virtualizace

1.1.1 Hardwarová a softwarová virtualizace

Základem virtualizace je software označovaný jako *Hypervisor* nebo *VMM* (Virtual Machine Monitor), který umožňuje vytvářet a spouštět jednotlivé virtuální stanice (VM, Virtual Machine). Stanice nebo server, na kterém je hypervizor spuštěn, je označována jako *hostitel* (host), a jednotlivé VM jsou označovány jako *hosté* (guests).

Podle klasifikace v [2] lze hypervizory rozdělit na dva typy:

1. Typ 1 (nativní).
   Nativní hypervizor (také označovaný jako *bare metal*) je spuštěný přímo na hardware hostitele. Nativní hypervizor přímo řídí všechny hardwarové zdroje hostitele a vystavuje je hostům, kteří jsou na něm spuštěni. Toto řešení bývá také označováno jako hardwarová virtualizace a umožňuje dosáhnout prakticky nejvyššího výkonu pro hostované VM, které mohou k hardwaru přistupovat přímo přes hypervizor. Mezi příklady nativních hypervizorů patří VMware ESXi, Microsoft Hyper-V, nebo Citrix XenServer.
2. Typ 2 (hostovaný)
   Hostovaný hypervizor je spuštěn na běžném operačním systému stejně jako ostatní programy. Toto řešení je také označováno jako softwarová virtualizace a má obvykle horší charakteristiky než nativní hypervizor na srovnatelném hardware, protože hypervizor přistupuje k hardware hostitele přes operační systém a nemůže jej řídit přímo. Zástupci tohoto typu hypervizoru jsou např. Oracle VirtualBox, VMware Player, VMware Workstation nebo QEMU.



Obr. 1.1: Rozdělení hypervizorů na typy[1]

Na obrázku 1.1 je zobrazeno rozdělení hypervizorů na zmiňované typy. Hranice mezi jednotlivými typy není ovšem v některých případech zcela jasná – např. hypervizor KVM je možné označit jako typ 1, protože je implementován formou modulů linuxového jádra, tudíž má přímý přístup k hardware, nicméně vzhledem k tomu, že

[1]Inspirováno [2]

v typickém linuxovém systému je spuštěna i řada dalších aplikací, které přistupují k hardwarovým zdrojům, lze ho také zařadit k ostatním hypervizorům typu 2.[3]

1.1.2 Virtualizace pracovních stanic

Virtualizace koncových stanic (desktop virtualization) je poměrně zajímavou alternativou nasazení koncových stanic. V praxi je používána zejména ve dvou variantách: lokální a vzdálená virtualizace.

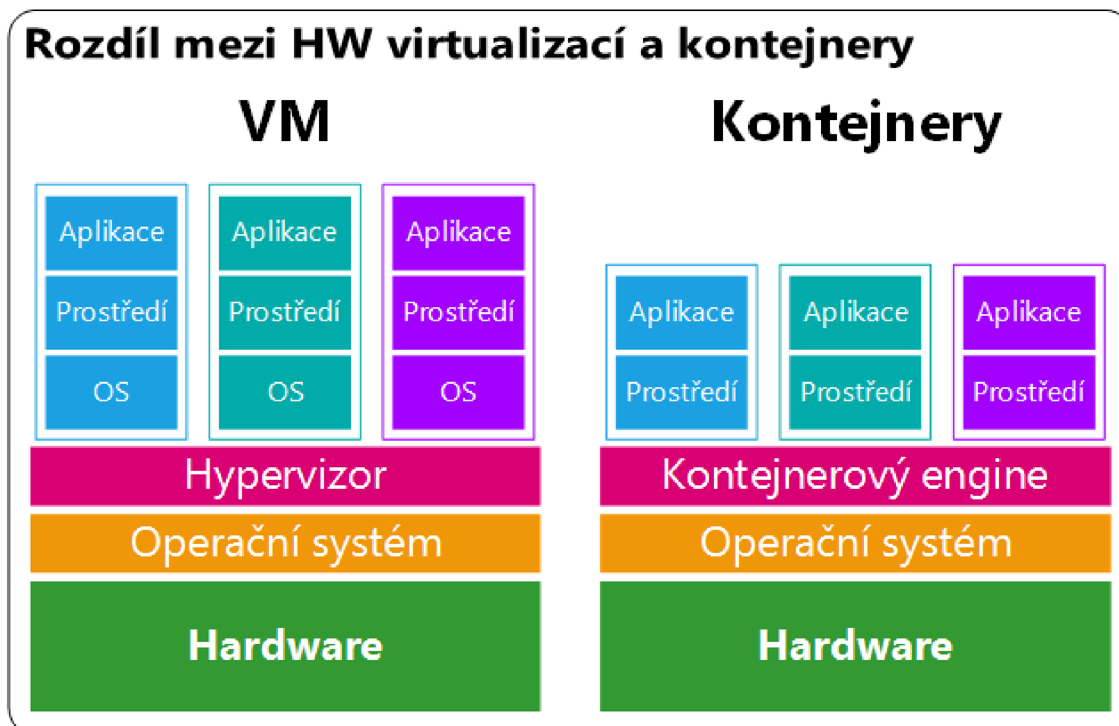
V případě lokální virtualizace je pracovní prostředí spuštěno přímo na koncové stanici pomocí hardwarové nebo softwarové virtualizace. Toto řešení lze uplatnit zejména v případech, kdy není možné realizovat nepřetržité spojení stanice s centrální infrastrukturou.

V případě vzdálené virtualizace se jedná o implementaci typu klient/server. Jednotlivé aplikace jsou spuštěny na vzdáleném operačním systému, přičemž uživatel s nimi komunikuje pomocí svého koncového zařízení. Tím může být běžné PC, tenký klient, případně i mobilní zařízení (tablet, smartphone). Veškerá data se nacházejí na centrálním serveru, a koncové zařízení slouží pouze k ovládání aplikací. Toto řešení bývá také označováno terminologií společnosti VMware jako VDI (Virtual Desktop Infrastructure). Zástupci této infrastruktury jsou např. řešení VMware View, Citrix XenApp, nebo služba RDS (Remote Desktop Services) integrovaná v rámci MS Windows.[4]

1.1.3 Virtualizace aplikací

Další variantou virtualizace je virtualizace aplikací, taktéž nazývaná jako *kontejnerová* virtualizace. Při tomto typu virtualizace se nepoužívají plnohodnotné virtuální stanice s vlastním operačním systémem, ale tzv. *kontejnery*, do kterých je daná aplikace uzavřena spolu se všemi svými závislostmi (knihovnami a jiným programovým prostředím), nutnými pro její běh. Tyto aplikace poté sdílí jádro (kernel) se systémem hostitele, ale jsou v jeho rámci různými mechanismy zcela izolovány od všech ostatních procesů spuštěných na hostiteli, ale také od procesů v ostatních kontejnerech. Na obrázku 1.2 je naznačen rozdíl mezi tradiční hardwarovou nebo softwarovou virtualizací a virtualizací aplikací.

Kontejnery mají na rozdíl od plnohodnotných VM výhodu v tom, že nemusí obsahovat celý operační systém, resp. vlastní jádro, protože jej sdílí s hostitelem. To se pozitivně promítá jednak do velikosti kontejneru, a jednak do režijních nákladů na jeho spuštění. Nevýhodou je fakt, že kontejnery musí být založeny na shodném OS, jako má hostitel, a dále neposkytují tak silnou izolaci od systému hostitele.[5]



Obr. 1.2: Rozdíl mezi HW/SW virtualizací a virtualizací aplikací[2]

Příkladem kontejnerové virtualizace jsou např. nástroje BSD Jails, Solaris Zones, OpenVZ, Linux-Vserver nebo Docker.

1.1.4 Ostatní typy virtualizace

Virtualizace sítí

Pod virtualizací sítě je možné si představit jak virtualizaci síťových zařízení (směrovačů, přepínačů, bran firewall ...) nebo částí sítě (sítě VLAN – fyzická síť je rozdělena na více logických segmentů), tak i virtualizaci celé síťové infrastruktury, kdy lze k síti přistupovat stejně, jako k VM, tzn. je možné automatizovaně vytvářet nebo měnit celé sítě. K tomuto řešení se často přistupuje v rámci architektury softwarově definovaných datových center (SDDC), kde je zcela virtualizován výpočetní výkon, úložiště, i síť. Další oblastí s velkým uplatněním virtualizovaných sítí jsou v současnosti různá cloudová řešení, jako je například OpenStack, Microsoft Azure nebo Amazon EC2, kdy je zákazníkům pronajímána výpočetní infrastruktura včetně možnosti konfigurace síťového prostředí dle jeho potřeb. [7]

[2]Převzato z [6]

Virtualizace úložišť

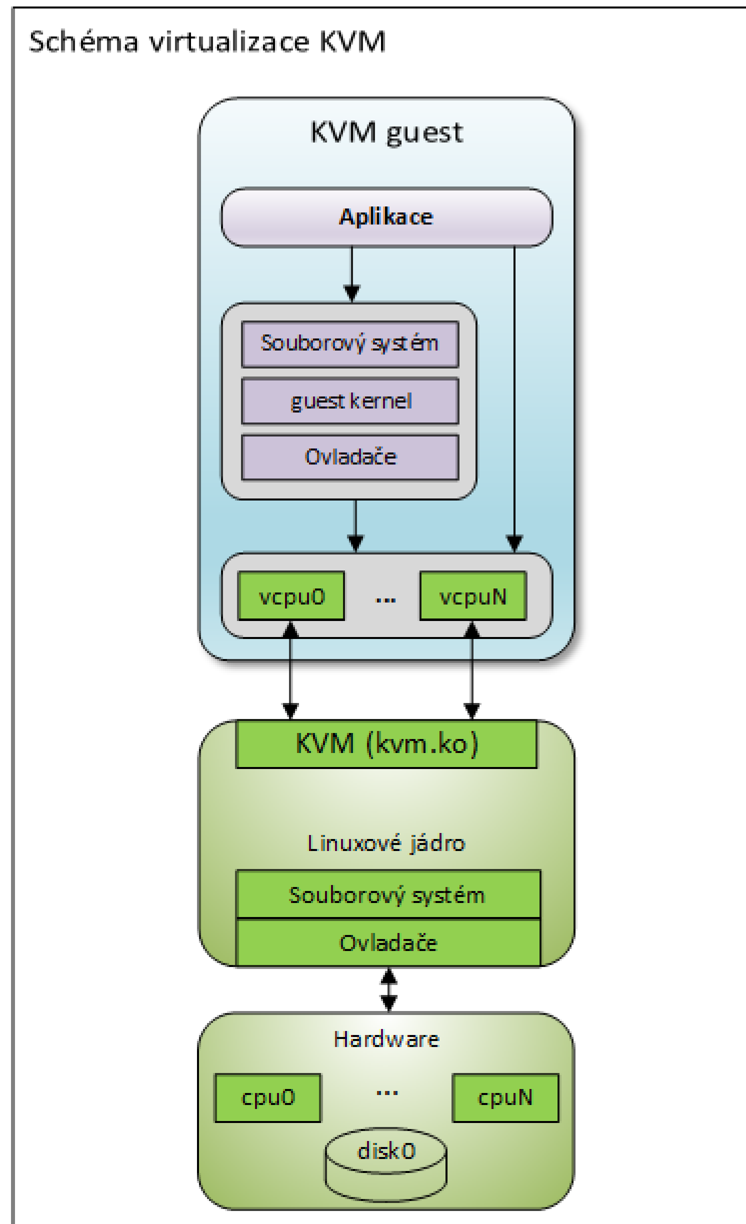
Jednou z možností, jak uspokojit nároky virtualizovaných stanic na diskový prostor jsou virtualizovaná úložiště. Nejčastěji je možné se setkat s řešením úložiště označovaným jako V-SAN (Virtual Storage Area Network), což je softwarově definované úložiště, které může zahrnovat velký počet fyzických disků nebo diskových polí napříč mnoha servery, a které je virtuálním stanicím prezentováno hypervizorem jako velký sdílený úložný prostor. Výhodou úložišť V-SAN je zjednodušení storage vrstvy (zdroje mohou být jednotlivým VM přiřazovány automaticky podle definovaných profilů, např. pro VM s vysokými nároky na I/O může být přiřazeno virtuální úložiště, které je definováno na rychlých discích SSD), vysoký výkon (V-SAN může např. obsahovat vyrovnávací paměti (cache) pro čtení a zápis ve formě SSD disků) a snížení nákladů (diskový prostor je možné rozdělovat mezi VM podle potřeby, a tak jej beze zbytků využít). [8]

1.1.5 Virtualizační metody použité při měření

KVM/QEMU a libvirt

KVM (Kernel-based Virtual Machine) je Open-source virtualizační řešení pro systém Linux. Skládá se z modulu jádra `kvm.ko`, který zajišťuje jádro virtualizační infrastruktury a modulu pro procesor (`kvm-intel.ko` nebo `kvm-amd.ko`). Procesor navíc musí podporovat příslušná rozšíření pro akceleraci virtualizace (`Intel VT-x` nebo `AMD-V`). KVM zpřístupňuje v systému hostitele (host) virtuální rozhraní `/dev/kvm`, které poté mohou využít uživatelské (user-space) aplikace ke správě paměti, I/O nebo dalších virtualizovaných komponent dané virtualizované stanice (guest). Jednou z těchto aplikací je např. QEMU, které využívá virtualizaci pomocí KVM, pokud je dostupná, čímž umožňuje dosáhnout výkonů srovnatelných s výkony bez použití virtualizace. Pokud KVM v systému dostupné není, může QEMU využít pouze softwarovou emulaci, která je ovšem méně výkonná. Na obrázku 1.3 je naznačeno schéma virtualizace KVM.

Tradičně se pro správu (nejen) prostředí KVM používá **libvirt**, což je rozsáhlé rozhraní API (Application Programming Interface), které slouží pro správu virtualizovaných instancí (v rámci libvirt nazývány jako `domains`) na hostitelské stanici (zde označováno jako `node`). Libvirt API zahrnuje velké množství funkcí pro správu domén od vytváření a upravování přes migraci až po monitorování až po funkce pro správu prostředků pro tyto domény – správu virtuálních sítí, úložišť, atd.[9] Všechny tyto funkce jsou k dispozici v jazyce C, včetně vazeb na další populární jazyky, např. Python. Pomocí libvirt API lze spravovat prakticky všechna oblíbená virtualizační prostředí: KVM/QEMU, Xen, LXC, OpenVZ, VirtualBox, VMware



Obr. 1.3: Schéma virtualizace KVM[3]

ESX a GSX, Microsoft Hyper-V, IBM PowerVM a mnohé další.[10]. Pomocí tohoto API je také realizováno velké množství známých nástrojů používaných pro správu virtualizovaného prostředí:[11]

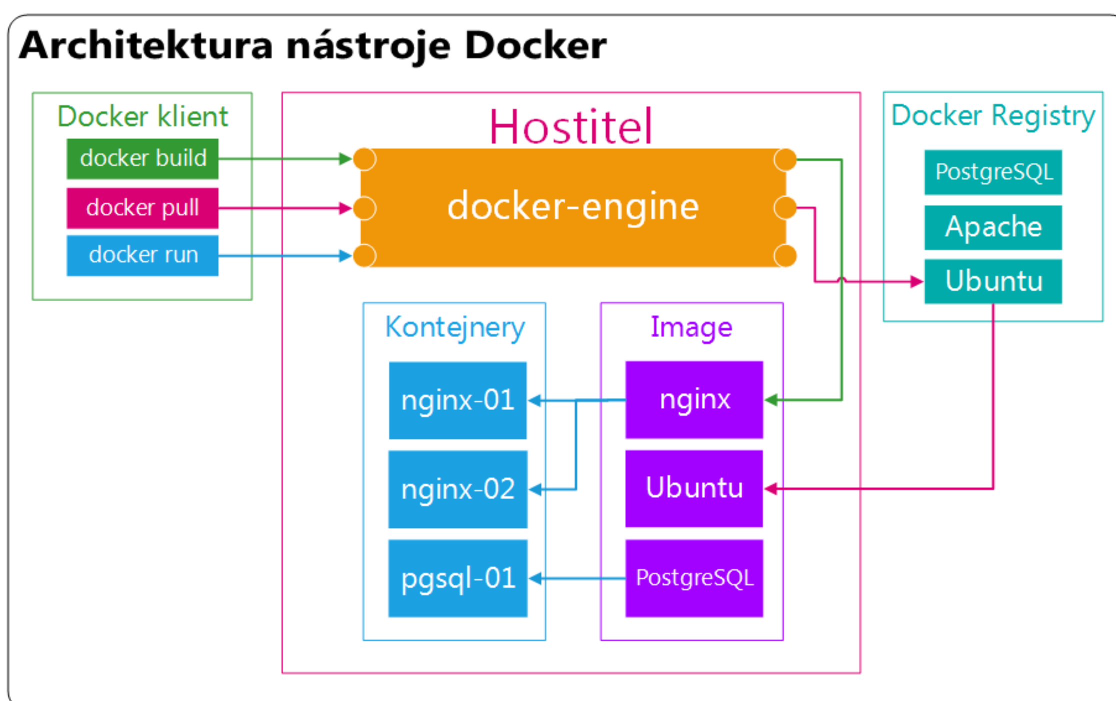
- `guestfish` – nástroj pro příkazovou řádku (CLI), sloužící pro správu a editaci virtuálních souborových systémů,
- `virsh` – CLI nástroj obsahující prakticky všechny funkce dostupné v rámci libvirt API, hojně používaný pro správu virtualizace na systémech v textovém režimu,

[3]Převzato z [12]

- `virt-install` – CLI nástroj pro automatizované vytváření nových VM,
- `virt-manager` – grafický nástroj pro správu virtualizace na lokálních nebo vzdálených hypervizorech,
- a další.

Docker

Docker je nástroj s otevřeným kódem určený primárně k usnadnění vývoje, údržby, dodávání a provozování aplikací pomocí aplikačních kontejnerů, které umožňují aplikaci izolovat v rámci kontejneru společně se všemi jejími závislostmi a programovým prostředím od ostatních procesů na systému hostitele. Na obrázku 1.4 je znázorněna architektura tohoto nástroje.



Obr. 1.4: Architektura nástroje Docker[4]

Nástroj Docker se skládá z několika částí:

- **Docker klient** – jedná se o nástroj příkazové řádky, napsaný v jazyce Go který slouží k ovládání samotného nástroje (`docker-engine`). Nástroj obsahuje množství příkazů pro správu kontejnerů a jejich obrazů (image), dále pro správu virtuálních úložišť a sítí nebo pro zobrazování různých informací. S nástrojem `docker-engine` komunikuje buď pomocí Unixového soketu (lokálně), nebo pomocí REST API (lokálně nebo vzdáleně).

[4]Převzato z [13]

- `docker-engine` – tento komponent nástroje je spuštěn na hostitelském systému a stará se o spouštění, zastavování a veškeré jiné operace s kontejnery nebo jejich obrazy. `docker-engine` přijímá příkazy od Docker klientů, které následně provádí.
- **`Docker registry`** – jedná se o repozitáře služby Docker (privátní nebo veřejné), ze kterých je možné stahovat obrazy kontejnerů (image), a případně je do nich také nahrávat. Nejznámější je *Docker Hub*, což je oficiální repozitář obsahující obrovské množství obrazů vycházejících z různých distribucí operačních systémů a poskytujících velké množství různých služeb a aplikací, populárních i neznámých.

Základem všech aplikačních kontejnerů je již zmíněná image, což je obraz souborového systému ve formátu **UnionFS**, který umožňuje vytvořit souborový systém postupným překrýváním transparentních vrstev. Tento přístup umožňuje sdílet mezi více obrazy disku stejné základní bloky, což ve výsledku šetří místo na disku a poskytuje mnoho dalších výhod, např. jednodušší aktualizace obrazů (stačí aktualizovat pouze ty vrstvy, ve kterých proběhly nějaké změny).

Image je možné získat buď stažením z Docker repozitářů, nebo vygenerováním pomocí tzv. *build* procesu ze souboru zvaného *Dockerfile*. Tyto soubory obsahují sekvenci příkazů, které se při vytváření image postupně provádějí a skládají tak na základní image další vrstvy, ze kterých je následně vytvořena výsledná image. Samotné aplikační kontejnery jsou poté spouštěny právě z těchto výsledných image.

Izolaci kontejnerů od hostitelského systému zajišťuje nástroj Docker pomocí tzv. *namespaces*, což je technologie linuxového jádra umožňující v rámci jednoho kernelu vytvořit více oddělených uživatelských prostorů (tzv. *userspace*). Nástroj také umožňuje řízené přidělování zdrojů pomocí tzv. *cgroups*, které jsou implementovány v rámci linuxového jádra a umožňují striktní kontrolu nad hardwarovými zdroji, jako je např. procesorový čas nebo velikost přidělené operační paměti. [5]

1.2 Protokoly a služby použité při měření

V následující části práce jsou stručně popsány jednotlivé síťové služby, pomocí kterých se přistupuje na virtuální stanice, a jejichž výkon a odezva jsou v rámci této práce měřeny pro virtualizační technologie Docker a KVM.

1.2.1 Protokol ICMP

Jedním z měřených parametrů virtuálních stanic je doba odezvy jejich síťového rozhraní, často také označována jako *latence* nebo zpoždění. Tato hodnota obecně vyjadřuje dobu, která uplyne mezi zasláním zprávy cílové stanici a obdržením její

odpovědi. Ke změření odezvy se používá standardní nástroj `ping`, který je dostupný prakticky na všech operačních systémech. Příkaz `ping` po spuštění zasílá cílové stanici pakety protokolu ICMP (tzv. `ECHO_REQUEST`), na které by cílová stanice měla odpovědět zprávou ICMP `ECHO_RESPONSE`, přičemž je následně vyhodnocena doba, která uplynula od zaslání výzvy a obdržení odpovědi (tzv. Round Trip Time, RTT) [14].

Protokol ICMP [5] je internetový protokol, který slouží k servisní komunikaci mezi stanicemi a výchozími bránami v sítích založených na IP protokolu, a každý, kdo komunikuje na IP vrstvě, jej musí povinně implementovat. ICMP zprávy se používají například pro odesílání informací stanici, že datagram, který od ní byl přijat, není možné zpracovat či přeposlat. Zprávy protokolu ICMP jsou zapouzdřeny do IP paketů s číslem protokolu 1. V rámci protokolu ICMP je definováno několik typů zpráv, přičemž právě typ určuje zbytek struktury záhlaví ICMP datagramu. Pro každý typ zprávy je pak definováno několik kódů zpráv. Mezi ICMP zprávy patří například tyto:

- `ECHO REPLY` – typ 0, kód 0, slouží k odpovědí na výzvu `ECHO REQUEST`.
- Cílová síť je nedostupná – typ 3, kód 0.
- Cílová stanice je nedostupná – typ 3, kód 1.
- Nutná fragmentace, ale je nastaven příznak Nefragmentovat (DF) – typ 3, kód 4.
- `ECHO REQUEST` – (typ 8, kód 0), žádost o zaslání odpovědi `ECHO REPLY`.
- Překročeno TTL[6] – typ 11, kód 0 [15].

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Typ      |      Kód      |        Kontrolní součet       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Identifikátor        |        Pořadové číslo         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Data ...
+-+-+-+-+-
```

Obr. 1.5: Záhlaví ICMP paketu[7]

Na obrázku 1.5 je zobrazen příklad záhlaví ICMP datagramu pro zprávy `ECHO REQUEST` a `ECHO REPLY`. V prvním případě je v poli typ uložena hodnota 8, ve druhém hodnota 0. Kód je u obou typů zpráv 0. Následuje kontrolní součet, což je jedničkový

[5] Internet Control Message Protocol.
[6] Time to Live, hodnota vyjadřující životnost paketu.
[7] Převzato z [15]

doplněk celého ICMP záhlaví. V poli identifikátor pak může být uložena vybraná hodnota, aby bylo možné snáze rozpoznat dvojice `REQUEST` a `REPLY` zpráv, které náleží k sobě. Pole s pořadovým číslem slouží ke stejnému účelu a je rovněž nepovinné (resp. může obsahovat nulovou hodnotu). Za ICMP hlavičkou následuje libovolné množství dat, která musí příjemce zprávy `ECHO REQUEST` zaslat zpět ve zprávě `ECHO REPLY` [15].

1.2.2 Protokol FTP

Dalším z protokolů použitých při měření parametrů virtualizovaných stanic je protokol FTP [8], který slouží pro přenos souborů mezi dvěma stanicemi. Protokol FTP pracuje v režimu klient/server a pro datové přenosy a přenos řídících dat používá dvě oddělená spojení. Protokol FTP umožňuje autentizaci pomocí uživatelského jména a hesla, nebo připojení k serveru v anonymním režimu. Protože jsou ale veškerá data přenášena nešifrovaně, při přenosu citlivých dat a souborů se doporučuje použít nástavbu FTPS, kde jsou přenosy zabezpečeny pomocí protokolů SSL a TLS.

FTP spojení je možné realizovat ve dvou režimech:

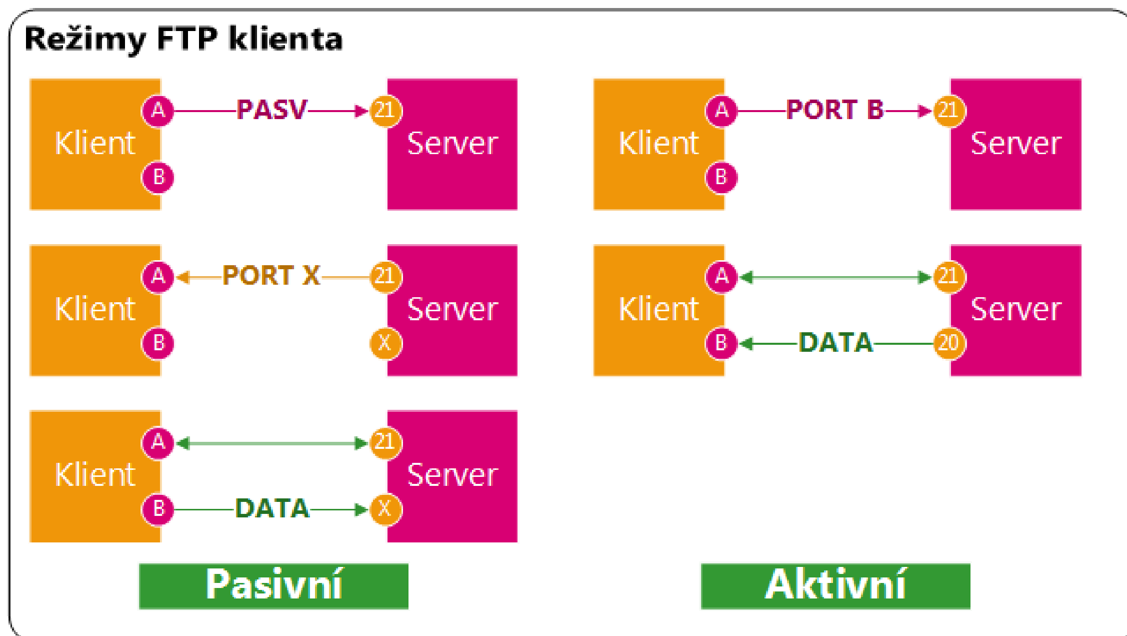
1. **Aktivní režim** V tomto režimu klient nejprve otevře řídící spojení protokolem TCP z náhodného portu na řídící port 21 FTP serveru. Přes toto spojení následně klient server informuje o portu, na kterém bude klient naslouchat příchozímu datovému spojení. Server poté vytvoří nové datové spojení ze svého portu 20 na port klienta, který mu byl oznámen.
2. **Pasivní režim** Tento režim je nutné použít v případě, že je před klientem brána Firewall, případně probíhá překlad adres NAT, v takovém případě totiž server nemůže vytvořit datové spojení směrem ke klientovi. V pasivním režimu nejprve klient otevře řídící spojení na port TCP 21 serveru stejně jako u aktivního režimu. Následně serveru příkazem `PASV` sdělí, že si přeje přejít do pasivního režimu. Odpovědí serveru je IP adresa a port, na kterém bude server naslouchat příchozímu datovému spojení. Klient následně vytvoří nové datové spojení ze svého náhodného portu na tento port a přenos dat může být zahájen[16].

Oba režimy spojení jsou ilustrovány na obrázku 1.6.

Samotné datové přenosy mohou být u protokolu FTP provedeny v několika režimech. Příkladem je režim `ASCII`, který se používá pro přenos textových souborů. V tomto režimu jsou data před přenosem konvertována do 8-bitové ASCII abecedy, pro jiné než textové soubory tedy není vhodný. Druhým z přenosových režimů je binární režim, kdy jsou data odesílána po bajtech bez další konverze.

Komunikace přes řídící spojení mezi klientem a serverem probíhá pomocí FTP příkazů a návratových kódů. Návratový kód je vždy (obdobně jako u protokolu

[8] File Transfer Protocol.



Obr. 1.6: Režimy přenosu FTP souborů[9]

HTTP) číslo složené ze tří číslic, z nichž první udává, zda návratový kód vyjadřuje kladnou či zápornou odpověď, a zda je tento stav pouze dočasný, nebo permanentní:

- `1xx` – Kladná odpověď před provedením akce. Tyto návratové kódy značí, že příkaz byl úspěšně přijat a že probíhá příprava k provedení požadované akce. Příkladem je kód 150, který udává, že požadovaný soubor je v pořádku a že bude zahájen datový přenos.
- `2xx` – Kladná odpověď po dokončení akce. Návratové kódy s touto hodnotou udávají, že požadovaná akce byla úspěšně dokončena a je možné zaslat další FTP příkaz (např. kód 250, operace se souborem proběhla úspěšně).
- `3xx` – Kladná odpověď během provádění akce. Tento kód značí, že ačkoliv byl předchozí příkaz akceptován, k dokončení požadované akce je třeba poskytnout další údaje. Typickým zástupcem této skupiny je kód 331, který je klientovi vrácen po zaslání uživatelského jména při přihlašování, a informuje klienta o tom, že server pro dokončení přihlášení očekává zaslání hesla.
- `4xx` – Dočasná záporná odpověď. Zasláním tohoto kódu server informuje klienta, že požadovaná akce nebyla provedena, ale že chybový stav, který ji způsobil, je dočasný, a je tedy možné pokusit se o provedení akce znova. Příkladem je kód 430, který informuje klienta o tom, že poskytnul chybné přihlašovací údaje a má přihlášení opakovat.
- `5xx` – Trvalá záporná odpověď. Příkaz byl zamítnut, požadovaná akce nebyla vykonána, a klient by se neměl pokoušet o opětovné zaslání stejného příkazu

[9]Inspirováno [16]

nebo sekvence, protože opět skončí s chybou. Do této skupiny patří například kód 501 – neplatná syntaxe příkazu[16].

Návratové kódy se ještě dále dělí do podskupin podle druhé číslice – např. podskupina `x0x` pro chyby syntaxe příkazů, podskupina `x2x` pro informace o spojení, nebo skupina `x3x` pro informace a chyby související s autentizací.

Na straně klienta je možné pro řízení spojení FTP serveru zasílat FTP příkazy. Mezi standardní FTP příkazy patří například:

- `DELE` – příkaz ke smazání cílového souboru,
- `NLST` – příkaz pro výpis souborů v cílovém adresáři,
- `PASV` – příkaz pro přechod do pasivního režimu,
- `USER` – příkaz identifikující uživatele při autentizaci,
- `PASS` – heslo pro autentizaci,
- `STOR` – příkaz pro uložení dat na server,
- `RETR` – příkaz pro stažení kopie dat ze serveru[17].

1.2.3 Služba SSH

V rámci měření parametrů virtualizačních nástrojů je jednou ze sledovaných hodnot také odezva služby SSH, která slouží ke vzdálenému připojení na operační systém. Obecně je SSH protokolem aplikační vrstvy modelů TCP/IP a ISO/OSI, který kromě vzdáleného přihlášení na cílový systém umožňuje např. i zabezpečený přenos souborů, případně zabezpečené spojení různých síťových služeb přes nezabezpečenou síť. Architektura služby SSH je stejně jako u ostatních služeb klient/server, kdy se aplikace v roli SSH klienta autentizuje proti vzdálenému SSH serveru. Protokol SSH byl navržen jako náhrada za starší službu pro vzdálený přístup Telnet, který je nezabezpečený. V současnosti specifikace rozlišuje dvě verze protokolu SSH, označované jako SSH-1 a SSH-2. Verze 2 obsahuje oproti starší verzi několik bezpečnostních vylepšení, jako je např. algoritmus výměny klíčů Diffie-Hellman nebo zajištění autentičnosti zpráv pomocí kódů MAC [18].

Protokol SSH je rozčleněn do tří vrstev, z nichž každá zastává specifickou funkci:

1. **transportní vrstva** – Tato vrstva se stará o sestavení zabezpečeného spojení, tj. o výměnu šifrovacích klíčů, autentizaci serveru, šifrování přenášených dat a kompresi. Vyšší vrstvy následně pomocí rozhraní této vrstvy přenášejí jinak nezabezpečená data bezpečným kanálem[19].
2. **autentizační vrstva** – V rámci této vrstvy probíhá autentizace klienta oproti vzdálenému serveru. Autentizační vrstva podporuje několik způsobů autentizace:
   - autentizace pomocí hesla,
   - autentizace pomocí veřejného klíče,

- interaktivní autentizace pomocí výzvy,
- autentizace pomocí rozhraní GSSAPI [20].

3. **vrstva spojení** – V rámci této vrstvy jsou definovány logické kanály společně s kanálovými a globálními žádostmi, které slouží pro poskytování služeb SSH. V rámci jednoho SSH spojení může být multiplexováno více logických kanálů, přičemž může ve všech probíhat obousměrná komunikace. Pomocí kanálových žádostí mohou být mezi koncovými body předávána data o kanálu, jako je např. velikost terminálového okna nebo návratový kód ukončeného procesu na serveru. Globální žádosti potom slouží např. k přesměrování portů [21].

1.2.4 Služba Telnet

Protokol Telnet je protokol aplikační vrstvy modelu ISO/OSI, který poskytuje obousměrné textově orientované spojení, typicky mezi dvěma virtuálními terminály. Uživatelská data jsou přenášena dohromady s řídícími daty 8-bitovým bajtově orientovaným datovým spojením pomocí protokolu transportní vrstvy TCP. Protokol Telnet byl původně navržen pro přenos dat mezi dvěma terminálovými zařízeními nebo procesy, kdy byly oba koncové body považovány za „síťový virtuální terminál“ (Network Virtual Terminal, NVT), což je abstraktní zařízení, které mělo představovat standard pro terminálová zařízení. Každý NVT se skládá ze zobrazovacího a vstupního zařízení, která zpracovávají příchozí, resp. generují odchozí data, v případě vstupního zařízení mohou být tato data „zrcadlena“ přímo na lokální zobrazovací zařízení (tzv. *echo* režim). Data mají být ukládána ve vyrovnávací paměti NVT až do doby, než bude dostupný jeden celý řádek dat, který bude následně zaslán protistraně nebo lokálně zpracován [22].

Protokol Telnet se nejčastěji používal pro vzdálený přístup na síťová zařízení, nicméně vzhledem k faktu, že se jedná o nezabezpečený protokol, který přenáší veškerá data jako prostý text, byl posléze nahrazen bezpečným protokolem SSH.

Na obrázku 1.7 je vidět část dat aktivního spojení protokolem Telnet zachycená síťovým analyzátorem Wireshark. Ve spodní části obrázku je možné vidět výzvu pro přihlášení ke vzdálenému systému. V dalších paketech je potom po jednotlivých znacích zachyceno uživatelské jméno použité pro přihlášení, včetně hesla. Všechny znaky jsou v zachycených paketech díky aktivnímu echo režimu dvakrát – každá stisknutá klávesa je odeslána na vzdálený server, který ji následně zašle zpět pro zobrazení na lokálním terminálu.

Službu Telnet lze také použít pro základní otestování některých síťových služeb, jako např. HTTP, FTP nebo SMTP serverů. Na ukázce 1.1 je uveden příklad ověření spojení s HTTP serverem právě pomocí služby Telnet.

```
25 0.04335 172.17.0.2    172.17.0.1    TCP     telnet > 54764 [ACK] Seq=98 Ack=114 Win=29056 Len=0 TSval=180513688
26 0.08575 172.17.0.2    172.17.0.1    TELNET  Telnet Data ...
27 0.12337 172.17.0.1    172.17.0.2    TCP     54764 > telnet [ACK] Seq=114 Ack=118 Win=29312 Len=0 TSval=18051370
28 1.35812 172.17.0.1    172.17.0.2    TELNET  Telnet Data ...
29 1.35816 172.17.0.2    172.17.0.1    TCP     telnet > 54764 [ACK] Seq=118 Ack=115 Win=29056 Len=0 TSval=18051401
30 1.35822 172.17.0.2    172.17.0.1    TELNET  Telnet Data ...
31 1.35823 172.17.0.1    172.17.0.2    TCP     54764 > telnet [ACK] Seq=115 Ack=119 Win=29312 Len=0 TSval=18051401
32 1.42202 172.17.0.1    172.17.0.2    TELNET  Telnet Data ...
33 1.42214 172.17.0.2    172.17.0.1    TELNET  Telnet Data ...
34 1.42215 172.17.0.1    172.17.0.2    TCP     54764 > telnet [ACK] Seq=116 Ack=120 Win=29312 Len=0 TSval=18051403
35 1.61410 172.17.0.1    172.17.0.2    TELNET  Telnet Data ...
▸ Frame 26: 86 bytes on wire (688 bits), 86 bytes captured (688 bits)
▸ Ethernet II, Src: 02:42:ac:11:00:02 (02:42:ac:11:00:02), Dst: 02:42:a7:db:34:fb (02:42:a7:db:34:fb)
▸ Internet Protocol Version 4, Src: 172.17.0.2 (172.17.0.2), Dst: 172.17.0.1 (172.17.0.1)
▸ Transmission Control Protocol, Src Port: telnet (23), Dst Port: 54764 (54764), Seq: 98, Ack: 114, Len: 20
▾ Telnet
    Data: b878d8d6daec login:
```

Obr. 1.7: Zachycená data protokolu Telnet

```
$ telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
test
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head><title>403 Forbidden</title></head><body>
<h1>Forbidden</h1>
<p>You dont have permission to access / on this server.</p><hr>
</body></html>
Connection closed by foreign host.
```

Ukázka 1.1: Otestování spojení s HTTP serverem pomocí nástroje Telnet

2 REALIZACE MĚŘÍCÍHO PRACOVIŠTĚ

V této části práce je popsána příprava měřícího pracoviště, resp. serveru, včetně všech nástrojů a služeb potřebných pro měření, a dále také realizace samotného měření, včetně jeho zhodnocení. Pro realizaci scénáře měření je potřebné zajistit a připravit následující:

- virtualizační server (*hypervizor/host*),
  - základní nastavení serveru,
  - balíčky virtualizačních nástrojů KVM a Docker,
  - balíčky ostatních nástrojů a služeb,
- virtualizované servery (KVM),
  - obraz disku,
  - konfigurace síťových parametrů,
  - instalace a konfigurace potřebných balíčků a služeb,
- kontejnery služby Docker,
  - šablona pro kontejnery (tzv. *Dockerfile*),
  - konfigurace parametrů služby Docker,
  - konfigurační soubory pro jednotlivé služby spuštěné v kontejnerech
- nástroje pro realizaci měření,
  - balíček připraveného nástroje `vte`,
  - prerekvizity pro spuštění nástroje `vte`.

2.1 Příprava prostředí KVM/QEMU

2.1.1 Instalace balíčků

Základem virtualizačního prostředí KVM jsou balíčky `libvirt`. Ty je do systému možné nainstalovat pomocí příkazu uvedeného na ukázce 2.1. Po restartu hypervi-

```
$ apt-get install qemu-kvm python-virtinst libvirt libvirt-python \
  libvirt-dev virt-manager libguestfs-tools python3-libvirt
```

Ukázka 2.1: Instalace virtualizačních balíčků

zoru nebo zavedení modulů libvirt a kvm do jádra příkazem `modprobe` je virtualizační prostředí připraveno. Pro správu virtualizačního prostředí slouží CLI nástroj `virsh`. Tento nástroj je velmi rozsáhlý a umožňuje spravovat a konfigurovat všechny aspekty prostředí KVM, např.:

- virtuální stanice (v KVM označované jako *domains*) – vytváření, úpravu konfigurace, provozní operace, migraci běžící VM na jiný server, apod.,

- parametry serveru – využití CPU a paměti, informace o provozním stavu,
- síťová konfigurace – vytváření a správa virtuálních sítí, rozhraní, včetně jejich adresace, případně různé síťové filtry apod.,
- konfigurace úložiště – od jednoduchých svazků (*volumes*), až po pole svazků nebo disků (*pools*),
- verzování a zálohování VM – vytváření jejich obrazů (*snapshots*).

Pro správu prostředí KVM je také dostupný grafický nástroj `virt-manager`, který lze použít na systémech s nainstalovaným grafickým prostředím. Pro komunikaci s moduly KVM používá stejně jako nástroj `virsh` knihovnu `libvirt`.

2.1.2 Konfigurace sítě

Dalším krokem přípravy prostředí KVM pro měření je konfigurace sítě. Po instalaci KVM by mělo být v systému k dispozici síťové rozhraní typu `bridge` (síťový most), většinou označené jako `virbr0`. Jak vyplývá z názvu tohoto rozhraní (zkráceně bridge, síťový most), pomocí tohoto rozhraní lze jednotlivým VM poskytnout přístup k dalším (např. vnějším) sítím. V případě potřeby je rozhraní možné vytvořit i ručně nástrojem `brctl`.

Prostředí KVM používá pro definici sítí konfigurační soubory ve formátu XML, které lze následně importovat pomocí nástroje `virsh`, konkrétně jedním z příkazů `virsh net-define` nebo `virsh net-create`, v závislosti na tom, zda má být síť perzistentní, nebo pouze dočasná. Protože ruční úprava souboru by byla pro větší počet VM velmi zdlouhavá, můžeme potřebné nastavení vygenerovat pomocí připraveného skriptu `generate_network_config.py`, jehož obsah lze vidět na ukázce 2.2.

Skript `generate_network_config.py` vygeneruje na základě zadaných parametrů (prefix MAC adresy, počet stanic, prefix názvu stanice) konfigurační soubor ve formátu XML. Jedná se o jednoduchý skript v jazyce Python, který používá třídu `cElementTree` z vestavěného (built-in) modulu `xml.etree` pro vytvoření XML stromu a jeho následnému zapsání v textové podobě do souboru `network.xml`.

Na ukázce 2.3 je možné vidět konfigurační soubor vygenerovaný pro pět stanic.

Vygenerovanou konfiguraci sítě poté importujeme do prostředí KVM příkazem `virsh net-create --file network.xml` a zajistíme její automatické spouštění příkazem `virsh net-autostart lab-network`. Pokud nyní při tvorbě nové VM nadefinujeme její virtuální síťové kartě jednu z MAC adres obsažených v konfiguraci sítě, DHCP server této VM následně přiřadí odpovídající adresu. Tím je zajištěna konzistence mezi názvem stanice, MAC adresou, a IP adresou pro všechny VM.

Aby měly jednotlivé VM přístup do sítě Internet, je nutné na serveru ještě povolit přeposílání paketů (forwarding). To zajistí parametr `net.ipv4.ip_forward=1` v souboru `/etc/sysctl.conf`. Pro funkční spojení je také potřeba překlad adres (NAT).

```python
#!/usr/bin/python
import xml.etree.cElementTree as et

MAC_BASE = 'de:ad:be:ef:00:'
NUM_ADDR = 253
NAME_PREF = 'centos-'

network = et.Element('network')
et.SubElement(network, 'name').text = 'lab-network'
et.SubElement(network, 'bridge', name='br0')
et.SubElement(network, 'forward')

ip = et.SubElement(network, 'ip', address='10.255.255.254', netmask = '255.255.255.0')
dhcp = et.SubElement(ip, 'dhcp')
et.SubElement(dhcp, 'range', start='10.255.255.1', end='10.255.255.253')

for i in range(1, NUM_ADDR + 1):
        mac = MAC_BASE + '{:02x}'.format(i)
        name = NAME_PREF + '{:03d}'.format(i)
        ip = '10.255.255.{}'.format(i)
        et.SubElement(dhcp, 'host', mac=mac, name=name, ip=ip)

tree = et.ElementTree(network)
tree.write('network.xml')
```

Ukázka 2.2: Skript `generate_network_config.py` pro automatické generování konfigurace sítě

Nastavení NAT je zajištěno automaticky nástroji knihovny `libvirt` při vytvoření virtuální sítě definováním potřebných pravidel ve službě `iptables`. Tato pravidla je možné ověřit příkazem `iptables -L -t nat`, který do konzole vypíše seznam aktivních pravidel v tabulce (ve službě `iptables` *chain*) `nat`.

2.1.3 Příprava výchozího obrazu disku – instalace systému

Protože cílem práce je provést virtualizaci operačního systému Linux, konkrétně distribuce CentOS, základem přípravy šablony, která bude následně použita jako výchozí bod pro všechny virtualizované servery, je stažení instalačních souborů této distribuce. Aby bylo možné dosáhnout maximálního počtu virtualizovaných serverů na jednom hypervizoru, byla zvolena varianta distribuce CentOS 6.5 Minimal. Instalační soubor (obraz disku ve formátu ISO) má velikost přibližně 400 MB a obsahuje pouze to nejnutnější pro spuštění a běh systému. Obraz disku lze stáhnout např. ze

```xml
<?xml version="1.0"?>
<network>
  <name>lab-network</name>
  <bridge name="br0"/>
  <forward/>
  <ip address="10.255.255.254" netmask="255.255.255.0">
    <dhcp>
      <range end="10.255.255.253" start="10.255.255.1"/>
      <host ip="10.255.255.1" mac="de:ad:de:ad:00:01" name="centos-001"/>
      <host ip="10.255.255.2" mac="de:ad:de:ad:00:02" name="centos-002"/>
      <host ip="10.255.255.3" mac="de:ad:de:ad:00:03" name="centos-003"/>
      <host ip="10.255.255.4" mac="de:ad:de:ad:00:04" name="centos-004"/>
      <host ip="10.255.255.5" mac="de:ad:de:ad:00:05" name="centos-005"/>
    </dhcp>
  </ip>
</network>
```

Ukázka 2.3: Vygenerovaná konfigurace sítě

serveru `http://vault.centos.org/6.5/isos/x86_64/` [1].

První virtuální server je již možné vytvořit přímo na KVM, např. pomocí nástroje `virt-install`.

```
$ virt-install \
    -n centos-00 \
    -r 512 --vcpus=1 \
    --os-type=linux \
    --graphics vnc,port=32001,keymap=local \
    --disk path=~/centos/centos-00.qcow2,format=qcow2,size=2 \
    --network=network=lab-network,mac=de:ad:be:ef:00:01,model=e1000 \
    --cdrom=/home/kral/CentOS-6.5-x86_64-minimal.iso
```

Ukázka 2.4: Použití nástroje `virt-install` pro vytvoření VM

Příkazem z ukázky 2.4 vytvoříme nový virtuální server s názvem `centos-00`. Tento server bude mít přiřazen jeden virtuální procesor, 512 MB operační paměti a výstup z jeho virtuální grafické karty bude dostupný na VNC serveru integrovaném v KVM na portu 32001. Dále bude pro tento server vytvořen obraz disku v zadaném umístění, ve formátu `qcow2` s maximální velikostí 2 GB, a bude mu přiřazena virtuální síťová karta se zadanou MAC adresou v definované síti. Server bude

[1] Dostupné k 30.11.2015

také obsahovat virtuální CD mechaniku s připojeným instalačním diskem ve formátu ISO.

K serveru by se nyní mělo být možné připojit pomocí libovolného VNC klienta (např. `ssvnc`), a provést instalaci systému standardním způsobem. Po dokončení instalace je možné se přihlásit do systému. V našem případě byl při instalaci vytvořen uživatel `centos` se stejným heslem, pro přihlášení tedy použijeme tyto údaje.

Protože systém CentOS v minimální konfiguraci obsahuje pouze minimum softwarových balíčků potřebných pro chod systému, je nyní nutné několik potřebných balíčků do systému doinstalovat. Příkazem

```
$ sudo yum install -y epel-release acpid nano wget telnet-server \
                      xinetd vsftpd supervisor
```

povolíme instalaci balíčků z repozitáře `EPEL` (Extra Packages for Enterprise Linux), ve kterém jsou dostupné některé z dalších balíčků, a následně nainstalujeme tyto balíčky:

- `acpid` – ACPI démon, slouží k notifikaci programů spuštěných v uživatelském režimu (user-space) na události ACPI, např. signálu pro vypnutí serveru,
- `nano` – textový editor,
- `wget` – nástroj pro stahování souborů pomocí protokolů HTTP, HTTPS a FTP,
- `telnet-server` – server služby telnet,
- `xinetd` – tzv. superserver, který zastřešuje servery ostatních síťových služeb,
- `vsftpd` – server služby FTP/SFTP,
- `supervisor` – nástroj, který spouští, spravuje a monitoruje definované služby spuštěné na serveru.

Dalším krokem je úprava konfiguračních souborů. Jako první upravíme konfigurační soubory `/etc/hosts` a `/etc/sysconfig/network`. Soubor `hosts` upravíme tak, aby obsahoval následující:

```
127.0.0.1   @HOSTNAME@
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
```

Cílový obsah souboru `network` bude potom:

```
NETWORKING=yes
HOSTNAME=@HOSTNAME@
```

V obou souborech byl nahrazen název stanice (`hostname`) za řetězec `@HOSTNAME@`, aby jej bylo možné snáze identifikovat a posléze automaticky nahradit pomocí skriptu.

Spuštěná virtuální stanice tedy nyní obsahuje minimální funkční systém spolu se všemi potřebnými balíčky aplikací a služeb, které budou využity při měření. V dalším kroku bude provedena konfigurace všech potřebných služeb, které budou následně testovány.

2.1.4 Konfigurace služeb

Jako první provedeme základní konfiguraci služby SSH. Server této služby je v systému již nainstalován, a ve výchozí konfiguraci umožňuje přihlášení pomocí uživatelského jména a hesla. Aby bylo možné se k serveru připojit vzdáleně bez zadávání hesla, použijeme autentizaci pomocí RSA klíče. To lze provést následující sekvencí příkazů uvedených na ukázce 2.5. Do vytvořeného souboru `/.ssh/authorized_keys` poté vložíme příslušný veřejný SSH klíč.

```
$ cd ~
$ mkdir .ssh
$ chmod 700 .ssh
$ touch .ssh/authorized_keys
$ chmod 644 .ssh/authorized_keys
```

Ukázka 2.5: Příprava pro přihlašování k SSH serveru pomocí klíče

Další ze služeb, kterou je třeba nakonfigurovat, je `telnet` server. Služba bude spouštěna v rámci superserveru `xinetd`, proto je potřeba upravit konfiguraci tohoto nástroje. Výchozí konfigurace se nachází v souboru `/etc/xinetd.conf`, a do konfigurace budou také zahrnuty všechny soubory z adresáře `/etc/xinetd.d`. Do tohoto adresáře také umístíme konfiguraci pro službu `telnet`, která je zobrazena na ukázce 2.6.

```
service telnet
{
        flags                   = REUSE
        socket_type            = stream
        wait                    = no
        user                    = root
        server                    = /usr/sbin/in.telnetd
        log_on_failure            += USERID
        disable                    = no
}
```

Ukázka 2.6: Konfigurace telnet serveru

Poslední z testovaných služeb je služba FTP. Jako server pro tuto službu bude použit `vsftpd` (*very secure ftp daemon*). Konfigurace je obsažena v souboru `vsftpd.conf` v adresáři `/etc/vsftpd/`, a měla by obsahovat položky, které jsou v ukázce 2.7.

```
anonymous_enable=NO
local_enable=YES
write_enable=YES
local_umask=022
dirmessage_enable=YES
xferlog_enable=YES
connect_from_port_20=YES
xferlog_file=/var/log/vsftpd.log
xferlog_std_format=YES
listen=YES
pam_service_name=vsftpd
userlist_enable=YES
tcp_wrappers=YES
```

Ukázka 2.7: Konfigurace FTP serveru

Konfigurace z ukázky 2.7 zajistí zejména možnost přihlášení lokálním uživatelům, včetně zápisu (direktivy `local_enable` a `write_enable`), dále zapnutí rozšířeného logování (direktivy `xferlog`), a také to, že bude server používat pouze protokol IPv4 (direktiva `listen`).

Aby bylo možné zajistit, že všechny zmíněné služby zůstanou v provozu po celou dobu testování, tedy i v případě, že dojde k neočekávané chybě, která by mohla zapříčinit pád jednoho ze serverů, resp. daemonů, použijeme nástroj, který bude služby periodicky monitorovat a zajistí jejich spuštění po startu serveru, a i v případě, že se služba neočekávaně zastaví. Takovým nástrojem je např. `supervisord`, který byl již nainstalován v předchozím kroku. Vzorovou konfiguraci tohoto nástroje je možné vygenerovat příkazem `echo_supervisord_conf > /etc/supervisord.conf`. Konfigurace obsahuje velké množství různých položek a možností, které v tomto případě lze ponechat na výchozích hodnotách a je rozdělena o sekcí, uvozených hranatými závorkami. Výchozími sekcemi jsou například `[supervisord]` a `[supervisorctl]`, které obsahují konfiguraci nástroje. Konfigurace služeb, které má nástroj spravovat, se poté umístí do samostatných sekcí (pro každou službu je nutné definovat vlastní sekci). Konfigurace pro použité služby je zobrazena na ukázce 2.8. Posledním krokem je zajištění automatického spouštění nástroje `supervisord` po startu, což lze provést zadáním příkazu `chkconfig supervisord on`.

V této fázi by měly být všechny služby na serveru nakonfigurovány a spuštěny a mělo by je tedy být možné otestovat klientskými programy jednotlivých služeb, tj. příkazy `ssh`, `telnet` a `ftp`.

```
[program:sshd]
command=/usr/sbin/sshd -D

[program:xinetd]
command=xinetd -dontfork

[program:vsftpd]
command=/usr/sbin/vsftpd /etc/vsftpd/vsftpd.conf
```

Ukázka 2.8: Část konfigurace nástroje `supervisord`

2.1.5 Finální úpravy obrazu disku

Nastavení serveru je tímto hotovo, a je možné přikročit ke konverzi obrazu disku na šablonu, která bude tvořit základ pro všechny virtualizované servery. Prvním krokem při přípravě výchozího obrazu disku je komprese stávajícího virtuálního disku. Pro disk byla původně alokována velikost 2 GB, avšak reálně je po instalaci systému a balíčků využito pouze cca 800 MB. Kompresi obrazu disku lze provést pomocí nástroje `qemu-img` zadáním příkazů z ukázky 2.9. Původní obraz disku nejprve přejmenujeme,

```
$ mv centos-00.qcow2 centos-00.qcow2.old
$ qemu-img convert -c -O qcow2 centos-00.qcow2.old centos-00.qcow2
```

Ukázka 2.9: Komprese obrazu disku pomocí nástroje `qemu-img`

a poté příkazem `qemu-img convert` provedeme konverzi. Parametr `-c` vynutí kompresi volného místa, parametr `-O` určuje výstupní formát konverze. V tomto případě je výstupní formát shodný se vstupním (`qcow2`).

Dalším krokem při přípravě šablony je uvedení obrazu disku do stavu, kdy nebude obsahovat žádnou specifickou konfiguraci ani soubory, které jsou nějakým způsobem vázány na VM, které byl disk původně přiřazen. K tomuto účelu lze využít nástroje `virt-sysprep`, který dekonfiguruje systém obsažený v obrazu disku a připraví obraz ke klonování. Nástroj umožňuje provádět velké množství operací, jejichž seznam lze zobrazit příkazem `virt-sysprep --list-operations`. Pro přípravu obrazu použijeme následující operace:

- `bash-history` – vymaže historii příkazů,
- `dhcp-client-state` – odstraní informace o adresách přidělených z DHCP,
- `net-hwaddr` – odstraní konfiguraci MAC adresy,
- `udev-persistent-net` – odstraní statickou konfiguraci parametrů sítě,
- `tmp-files` – vymaže dočasné soubory.

Celý příkaz je uveden na ukázce 2.10.

```
$ virt-sysprep --enable bash-history,dhcp-client-state,net-hwaddr,\
  udev-persistent-net,tmp-files -a centos-00.qcow2
```

Ukázka 2.10: Použití nástroje `virt-sysprep`

Posledním krokem přípravy šablony je zajištění vhodného názvu pro stanici (`hostname`), který by měl být pro každou VM unikátní, ale navíc generovaný podle určitých pravidel – pro snazší orientaci by bylo vhodné, aby se z `hostname` dala odvodit IP a MAC adresa a obráceně. K tomu slouží skript `set_hostname.sh`, jehož obsah je na ukázce 2.11.

```
#!/bin/bash
# set hostname based on primary interface MAC address
sed -i "s/@HOSTNAME@/centos-‘ifconfig eth0 | awk ’/HWaddr/ {print $5}’ | \
cut -d ’:’ -f 6‘/g" /etc/hosts /etc/sysconfig/network
# reboot just to be safe
reboot
```

Ukázka 2.11: Skript `set_hostname.sh`

Skript `set_hostname.sh` nahradí řetězec `@HOSTNAME@` v příslušných konfiguračních souborech za kombinaci řetězce `centos-` a posledního oktetu MAC adresy, kterou pomocí nástrojů `awk` a `cut` vyextrahuje z výstupu příkazu `ifconfig`. Skript dále restartuje VM příkazem `reboot`.

Spuštění skriptu `set_hostname.sh` lze vyvolat při prvním startu VM opět pomocí nástroje `virt-install`, konkrétně operací `firstboot`, jak je uvedeno na ukázce 2.12.

```
$ virt-sysprep --enable firstboot --firstboot \
  "/home/kral/backup-conf/set_hostname.sh" -a centos-00.qcow2
```

Ukázka 2.12: Použití nástroje `virt-sysprep` pro jednorázové spuštění skriptu.

Tímto krokem je příprava výchozího obrazu disku dokončena. Disk je nyní možné naklonovat a použít pro instalaci libovolného počtu unikátních VM.

2.1.6 Instalace virtuálních stanic

Z výchozího diskového obrazu můžeme nyní vygenerovat libovolný počet virtuálních stanic. Ty je možné vytvořit například pomocí nástroje `virt-install`. Protože pro testování bude zapotřebí velký počet virtuálních stanic, bylo by jejich ruční vytváření

příliš zdlouhavé. Z toho důvodu využijeme pro instalaci cílového počtu stanic skript, který je uveden na ukázce 2.13.

```
#!/usr/bin/env python
import subprocess
import time

PATH = '/srv/'

for i in range(1, 151):
    args = [
        'virt-install',
        '-n centos-{:03d}'.format(i),
        '-r 512',
        '--vcpus=1',
        '--os-type=linux',
        '--vnc',
        '--vncport=32{:03d}'.format(i),
        '--disk path={}centos-{:03d}.qcow2,format=qcow2'.format(PATH, i),
        '--network=network=net-kvm,mac=de:ad:be:ef:00:{:02x}'.format(i),
        '--import'
    ]
    cmd = ' '.join(args)
    print cmd
    p = subprocess.Popen(cmd, shell=True)
    p.communicate()
    time.sleep(10)
```

Ukázka 2.13: Skript `vm_install.py` pro automatické vytvoření zadaného počtu VM

Skript `vm_install.py` z ukázky 2.13 opakovaně volá příkaz `virt-install` s definovanými parametry, a mezi jednotlivými voláními vyčká 10 sekund, aby nebyla naráz vygenerována přílišná zátěž na systém, protože během prvního startu VM se jednorázově provádí přípravné operace popsané v předchozích sekcích a VM se navíc ještě restartuje.

Příkaz `virt-install` je volán s následujícími parametry:

- `-n centos-{:03d}` – jméno VM, řetězec ve složených závorkách bude nahrazen za trojmístné celé číslo,
- `-r 512` – VM bude přiděleno 512 MB virtuální paměti RAM,
- `--vcpus 1` – VM bude přidělen jeden virtuální procesor,
- `--os-type=linux` – informativní parametr o typu OS,
- `--vnc` a `--vncport` – grafický výstup z VM bude dostupný na VNC serveru KVM na daném portu,
- `--disk path, format` – cesta k připravenému obrazu disku,

- `--network net, mac` – definice síťových parametrů VM,
- `--import` – určuje, že importujeme do KVM již vytvořený obraz disku.

Po dokončení skriptu by měl být v systému vytvořen zadaný počet VM, což lze ověřit např. zadáním příkazu `virsh list --all`, který vypíše seznam všech definovaných (aktivních i neaktivních) VM.

2.2 Příprava prostředí Docker

V rámci této části práce bude popsána, obdobně, jako v sekci 2.1, příprava prostředí pro spouštění virtuálních kontejnerů služby Docker.

2.2.1 Instalace balíčků

Prvním krokem při instalaci balíčků služby `Docker` je ověření, zda jsou splněny všechny podmínky nutné pro běh této služby. Pro běh služby je nutný 64-bitový linuxový systém, který používá jádro (kernel) verze 3.10, nebo vyšší. Verzi jádra je možné ověřit například pomocí příkazu `uname -r`, který verzi jádra systému vypíše do konzoly.

Dalším krokem je přidání oficiálního repozitáře služby `Docker`. Balíčky jsou sice dostupné i v repozitářích linuxových distribucí, nicméně většinou se jedná o příliš staré verze. Zadáním příkazů z ukázky 2.14 se nejprve ujistíme, že balíčkovací systém `apt` má podporu pro stahování balíčků přes HTTPS, následně přidáme do systému GPG klíč pro ověření repozitáře, obnovíme seznam balíčků a konečně nainstalujeme balíček `docker-engine`. Před instalací je ještě možné pomocí příkazu `apt-cache policy docker-engine` ověřit, že bude balíček opravdu nainstalován z oficiálního Docker repozitáře.

```
$ apt-get install apt-transport-https ca-certificates
$ apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 \
  --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
$ echo 'deb https://apt.dockerproject.org/repo debian-jessie main '\
  >> /etc/apt/sources.list.d/docker.list
$ apt-get update
$ apt-get install docker-engine
```

Ukázka 2.14: Instalace balíčků Docker

Funkčnost nástroje Docker si po instalaci můžeme ověřit pomocí příkazu `sudo docker run hello-world`. Tento příkaz stáhne ze serveru Docker-Hub obraz kontejneru `hello-world` a následně ho spustí. Po zadání příkazu by se měl do terminálu

vypsat podobný obsah, jako na ukázce 2.15.

```
$ dockr run hello-world
  Unable to find image 'hello-world:latest' locally
  latest: Pulling from library/hello-world

  03f4658f8b78: Pull complete
  a3ed95caeb02: Pull complete
  Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
  Status: Downloaded newer image for hello-world:latest

  Hello from Docker.
  This message shows that your installation appears to be working correctly.
```

Ukázka 2.15: Ověření funkčnosti nástroje Docker pomocí předpřipraveného kontejneru

Nástroj `Docker` je tedy funkční a můžeme přikročit k jeho konfiguraci.

2.2.2 Konfigurace

Protože příkazy nástroje Docker je nutné spouštět s oprávněním superuživatele (`root`), což je při větším množství příkazů nepohodlné, můžeme uživatele přidat do systémové skupiny `docker`, která byla vytvořena při instalaci balíčků, čímž tento uživatel získá možnost nástroj Docker spravovat. To lze provést např. pomocí příkazu `usermod -aG docker <uživatel>`. Po odhlášení a opětovném přihlášení by měl tento uživatel mít možnost využívat veškeré dostupné příkazy nástroje Docker, což lze ověřit např. zadáním `docker info` do terminálu, přičemž by se následně měly vypsat základní informace o systému a instalaci nástroje.

Další konfigurací, kterou je potřeba upravit, jsou startovací parametry Docker démona. Ten ve výchozím nastavení naslouchá příkazům přes unixový soket v adresáři `/var/run`. Toto nastavení je ale potřeba změnit, protože knihovna pro komunikaci s REST API, kterou používá nástroj `vte`, popsaný v kapitole 2.3, neumí bez pluginu přes unixový soket komunikovat. Dalším parametrem, který je třeba nastavit, je adresa HTTP proxy, protože systém, na kterém budou testy provozovány, nemá přímé spojení do internetu. Obě zmíněné možnosti lze Docker démonovi předat již při startu přidáním patřičných parametrů do konfiguračního souboru. U linuxových distribucí, které používají pro start inicializační systémy `Upstart`, nebo `SysVinit`, se konfigurace nachází v souboru `/etc/default/docker`. U novějších verzí distribuce Debian je ovšem již použit `systemd`. U tohoto inicializačního systému je konfiguraci Docker démonovi možné předat vytvořením speciálního souboru `/etc/systemd/system/docker.service`, jehož obsah je na ukázce 2.16.

```
[Service]
Environment="HTTP_PROXY=<adresa proxy>"
ExecStart=
ExecStart=/usr/bin/docker daemon -H tcp://
```

Ukázka 2.16: Konfigurace parametrů Docker démona

Touto konfigurací je zajištěno, že bude Docker démon komunikovat přes zadaný HTTP proxy server, a že jeho REST API bude dostupné na adrese 127.0.0.1 a výchozím TCP portu 2375.

2.2.3 Příprava image

Stejně jako v případě KVM/QEMU, i u nástroje Docker je nutné připravit základní obraz disku, resp. kontejner (zde nazývaný `image`), ze kterého bude následně vygenerován potřebný počet kontejnerů pro testy.

Na rozdíl od KVM, kde bylo nutné systém ručně nainstalovat včetně všech potřebných balíčků a následně nakonfigurovat, aby jej bylo možné použít jako základ pro větší množství virtuálních stanic, v případě Dockeru probíhá příprava výchozího kontejneru/image plně automaticky na základě předem definované sekvence příkazů ve formě souboru, tzv. `Dockerfile`. `Dockerfile` může obsahovat libovolný počet instrukcí, přičemž při procesu přípravy image je následně každá instrukce převedena do vrstvy (layer). Vrstvy společně s výchozí image potom tvoří výslednou image, ze které je následně možné vytvořit libovolný počet kontejnerů. První část souboru `Dockerfile` je vidět na ukázce 2.17.

```
FROM centos:latest
MAINTAINER xkralj08@stud.feec.vutbr.cz
RUN yum install -y openssh-server telnet-server telnet vsftpd \
    xinetd nano lsof epel-release

RUN yum install -y supervisor
...
```

Ukázka 2.17: První část souboru Dockerfile

V první části souboru `Dockerfile` je uvedeno několik instrukcí:

- `FROM` – Instrukce `FROM` udává, která image bude použita jako základ pro všechny další vrstvy. Tato instrukce je povinná a musí být v rámci platného Dockerfile uvedena vždy jako první.
- `MAINTAINER` – Tato instrukce slouží pro identifikaci autora dané image.

- `RUN` – Instrukce `RUN` slouží k provedení příkazu v rámci vrstvy kontejneru, v tomto případě například pro instalaci potřebných balíčků pomocí správce balíčků `yum`. Jedná se o identické balíčky jako při přípravy obrazu disku u KVM, viz sekce 2.1.3.

Po dokončení instrukcí uvedených na ukázce 2.17 obsahuje image nainstalovaný systém a potřebné balíčky, které je potřeba nakonfigurovat. Protože proces přípravy image je navržen tak, aby byl bezobslužný, je nutné mít konfigurační soubory již dopředu připraveny. Protože konfigurace Docker kontejneru a virtuální stanice KVM/QEMU je identická, můžeme použít již připravené soubory, které do image nakopírujeme instrukcemi uvedenými na ukázce 2.18.

```
...
RUN useradd -mU test && echo "test:test" | chpasswd
# setup ssh
ADD ssh.tar /tmp
ADD id_rsa_test.pub /home/test/.ssh/authorized_keys
RUN mv -f /tmp/ssh/* /etc/ssh/ && \
    chmod 700 /home/test/.ssh && \
    chmod 600 /home/test/.ssh/authorized_keys && \
    chown -R test:test /home/test
...
```

Ukázka 2.18: Druhá část souboru Dockerfile

Kromě instrukcí `RUN` obsahuje část uvedená na ukázce 2.18 také instrukci `ADD`, jejímiž parametry jsou zdrojový soubor na disku a cílové umístění v rámci kontejneru. V tomto případě je soubor `ssh.tar`, který obsahuje konfigurační soubory SSH a předgenerované veřejné a soukromé RSA klíče, přenesen do umístění `/tmp` v rámci kontejneru. Archivy jsou navíc během kopírování automaticky extrahovány do cílového umístění, proto jej není potřeba zvlášť rozbalovat a můžeme jej rovnou pomocí příkazu `mv` přesunout do cílového umístění. Příkazy na ukázce dále nastaví příslušná oprávnění složkám obsahujícím SSH klíče a soubor `authorized_keys`, a také do systému přidá uživatele `test` se stejným heslem. Poslední část souboru je potom uvedena na ukázce 2.19.

Kromě již známých instrukcí `RUN` a `ADD` je zde také instrukce `CMD`. Ta určuje spustitelný soubor nebo příkaz, který bude vykonán po startu kontejneru. Protože je na tento příkaz navázán i celý životní cyklus kontejneru, je nutné zvolit takový příkaz, který bude permanentně běžet v popředí (foreground), v opačném případě se kontejner po doběhnutí příkazu ukončí. Pokud bychom například chtěli v kontejneru provozovat SSH server, je nutné jej spustit s parametrem `-D`, který zajistí, že server zůstane běžet v popředí. V tomto případě kontejner spustí příkaz `supervisord`, což

```
...
# setup telnet
ADD telnet /etc/xinetd.d/telnet
ADD xinetd.conf /etc/xinetd.conf

# setup ftp
ADD vsftpd.conf /etc/vsftpd/vsftpd.conf
RUN cat /dev/urandom | head -c 10M > /home/test/download.bin

# setup supervisord
ADD supervisord.conf /etc/supervisord.conf

CMD ["/usr/bin/supervisord", "-c", "/etc/supervisord.conf"]
```

Ukázka 2.19: Závěrečná část souboru Dockerfile

je již známý správce procesů/démonů, který byl již popsán v kapitole 2.1.3. Zbylé příkazy z této ukázky zajistí zkopírování konfiguračních souborů pro služby Telnet a FTP do kontejneru, a také vygenerování souboru o velikosti 10 MB z náhodných dat získaných z `/dev/urandom`, který bude použit jako testovací soubor při měření rychlosti stahování dat protokolem FTP.

2.2.4 Vytvoření image

Z připraveného souboru `Dockerfile` je nyní možné vytvořit image, která bude následně sloužit jako obraz souborového systému pro všechny kontejnery z ní vytvořené. Přejdeme tedy do adresáře, ve kterém se nachází soubor Dockerfile a příkazem `docker build --tag kral/centos-all ./` zahájíme proces tvorby image. Parametr `tag` slouží k označení image. V případě, že je `build` proces prováděný na serveru, který nemá přístup k internetu, resp. využívá HTTP proxy, je nutné procesu tuto informaci předat pomocí parametru `--build-arg=http_proxy=<adresa proxy>`, v opačném případě by proces při instalaci balíčků do kontejneru selhal, protože by neměl přístup k repozitářům.

Po zadání příkazu `docker build` se spustí proces vytvoření image, jehož část je na ukázce 2.20.

Z výpisu je možné vidět, že vytvoření image probíhá po jednotlivých vrstvách, které společně tvoří celkovou image. Protože bylo v `Dockerfile` 14 instrukcí, výsledná image bude mít 14 vrstev. Na ukázce 2.20 je zobrazena část vykonaných kroků, konkrétně lze vidět první tři instrukce, během kterých je definována výchozí image, a poté jsou nainstalovány balíčky. Příkazy jsou vykonávány v tzv. mezilehlých (intermediate) kontejnerech, ze kterých se následně vygeneruje daná vrstva image, a tento

```
Sending build context to Docker daemon 302.6 kB
Step 1 : FROM centos:latest
 ---> 0f0be3675ebb
Step 2 : MAINTAINER jan.kral@itself.cz
 ---> Using cache
 ---> 815815e377b9
Step 3 : RUN yum install -y   openssh-server   telnet-server   telnet   vsftpd   \
             xinetd nano   lsof   epel-release
 ---> Running in 2fbee4f49dae
Loaded plugins: fastestmirror, ovl
Determining fastest mirrors
Resolving Dependencies
--> Running transaction check
---> Package epel-release.noarch 0:7-5 will be installed
...
Step 12 : RUN cat /dev/urandom | head -c 10M > /home/test/download.bin
 ---> Running in 7d550592300c
 ---> d2e320c1108e
Removing intermediate container 7d550592300c
Step 13 : ADD supervisord.conf /etc/supervisord.conf
 ---> 5ce471957dac
Removing intermediate container d90db1dc3798
Step 14 : CMD /usr/bin/supervisord -c /etc/supervisord.conf
 ---> Running in e81047359b21
 ---> b5535b2cfc8b
Removing intermediate container e81047359b21
Successfully built b5535b2cfc8b
```

Ukázka 2.20: Proces vytvoření image pomocí příkazu `docker build`

mezilehlý kontejner je následně odstraněn (např. výpis pod krokem 12). Výsledkem `build` procesu je potom finální image, které je zde přiřazeno ID `b5535b2cfc8b`, lze s ní také ale pracovat pod názvem, který jí byl přidělen v minulém kroku (tag `kral/centos-all`). Zadáním příkazu `docker images` si potom můžeme zobrazit seznam všech obrazů kontejnerů dostupných v systému, v tomto případě máme pouze jednu image, jak je vidět na ukázce 2.21.

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
kral/centos-all     latest              b5535b2cfc8b        9 hours ago         368.7 MB
```

Ukázka 2.21: Seznam dostupných image

Pro danou image si lze také kdykoliv zobrazit všechny její detaily (příkaz `docker inspect <id>`, nebo přehledný seznam všech vrstev, ze kterých je image vytvořena, včetně jednotlivých instrukcí, ze kterých vrstvy vzešly, včetně jejich velikostí. K tomu

slouží příkaz `docker history <id>`, jehož výstup si můžeme prohlédnout na ukázce 2.22.

```
IMAGE          CREATED        CREATED BY                                      SIZE
b5535b2cfc8b   9 hours ago    /bin/sh -c #(nop) CMD ["/usr/bin/supervisord"   0 B
5ce471957dac   9 hours ago    /bin/sh -c #(nop) ADD file:a47e9ec5b44d8d082d   5.156 kB
d2e320c1108e   9 hours ago    /bin/sh -c cat /dev/urandom | head -c 10M > /   10.49 MB
5a2795ca88a5   9 hours ago    /bin/sh -c #(nop) ADD file:727aa5ce7bbefbdd2e   353 B
89c40cff84a1   9 hours ago    /bin/sh -c #(nop) ADD file:219d1418bc488f06ee   1.013 kB
7e75b1ba7795   9 hours ago    /bin/sh -c #(nop) ADD file:a500d10fede142d018   156 B
1255a3a7ff4f   9 hours ago    /bin/sh -c mv -f /tmp/ssh/* /etc/ssh/ &&        251.7 kB
926fd952b25d   9 hours ago    /bin/sh -c #(nop) ADD file:99b0522bf3ace2947f   735 B
33f052e7d1b9   9 hours ago    /bin/sh -c #(nop) ADD file:f34414a695c77cc921   250.5 kB
fb8f51690554   9 hours ago    /bin/sh -c useradd -mU test && echo "test:tes   4.74 kB
a2425b470354   9 hours ago    /bin/sh -c yum install -y supervisor            49.67 MB
a1f753bd230a   9 hours ago    /bin/sh -c yum install -y   openssh-server      111.4 MB
815815e377b9   5 weeks ago    /bin/sh -c #(nop) MAINTAINER jan.kral@itself.   0 B
0f0be3675ebb   7 weeks ago    /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>      7 weeks ago    /bin/sh -c #(nop) LABEL name=CentOS Base Imag   0 B
<missing>      7 weeks ago    /bin/sh -c #(nop) ADD file:6389065673f90a5cb6   196.6 MB
<missing>      7 months ago   /bin/sh -c #(nop) MAINTAINER The CentOS Proje   0 B
```

Ukázka 2.22: Seznam vrstev image `kral/centos-all`

Na zmíněné ukázce 2.22 můžeme vidět velikost jednotlivých vrstev image, v tomto případě jsou největšími vrstvami výchozí image distribuce CentOS (196,6 MB) a image, ve kterých došlo k instalaci balíčků (image `a1f753bd230a` s 111,4 MB a image `a2425b470354` o velikosti 49,67 MB). Součet velikosti všech vrstev potom udává celkovou velikost výsledné image, jak je uvedena na ukázce 2.21 (368,7 MB). Výhodou nástroje Docker je, že pokud bychom chtěli vytvořit další image, která by se lišila např. pouze v poslední instrukci (spouštěném příkazu), tato image by na disku nezabrala téměř žádné místo, protože by se použily již jednou vytvořené vrstvy. Image je tedy vhodné navrhovat tak, aby sdílely pokud možno co nejvíce instrukcí.

2.2.5 Instalace kontejnerů

Po přípravě image je již vytvoření kontejnerů jednoduché. Nový kontejner lze vytvořit příkazem `docker run`, ale protože je nutné vytvořit větší množství kontejnerů, zapouzdříme volání příkazu do `for` cyklu. Na ukázce 2.23 je potom zobrazen celý příkaz, který lze zadat do terminálu.

Na ukázce 2.23 je příkaz `docker run` volán s následujícími parametry:

- `-d` – Spustí kontejner v tzv. odpojeném (`detached`) módu, tzn. kontejner je spuštěn na pozadí a do konzoly je vypsáno ID nově vytvořeného kontejneru.

```
$ for i in $(seq 1 150);
  do
    docker run -d --name all-$i --label 'service'='vte' kral/centos-all;
  done
```

Ukázka 2.23: Vytvoření kontejnerů z image `kral/centos-all`

Bez zadání tohoto parametru je kontejner spuštěn v popředí s připojeným terminálem (`tty`).

- `--name all-$i` – Udává jméno (`alias`), kterým bude kontejner označen. Každý kontejner má svoje unikátní ID, které s používá při operacích právě s tímto kontejnerem, nicméně v případě, že má kontejner definované také jméno, je možné pro identifikaci kontejneru použít právě jméno namísto ID. V tomto případě bude název všech kontejnerů obsahovat řetězec `all` společně s číslem z rozsahu 0-150.
- `--label` – Parametr `label` umožňuje kontejneru přiřadit metadata, na základě kterých je možné jej identifikovat. Metadata jsou definována ve formátu `klíč = hodnota`. V tomto případě budou mít všechny vytvořené kontejnery v poli `service` hodnotu `vte`, na základě které je potom možné při výběru aktivních kontejnerů vybrat pouze ty kontejnery, které byly vytvořeny speciálně pro měření.
- `kral/centos-all` – Posledním parametrem příkazu `docker run` je image, ze které má být kontejner vytvořen. Parametr se uvádí ve formátu `[repozitář] / image[:značka]`, a je povinný. Značkou bývá většinou číslo verze image, např. `1.0` nebo `latest`, v případě, že není uvedena, se automaticky vybere image s nejvyšší verzí (`latest`).

2.3 Vytvořená aplikace `vte`

Pro potřeby automatického měření vybraných parametrů virtuálních stanic byl vytvořen nástroj s názvem `vte`, což je zkratka pro `Virtualization Testing Environment` (prostředí pro testování virtualizace). Nástroj `vte` umožňuje v aktuální implementaci automaticky provádět testování několika vybraných služeb – síťovou odezvu pomocí nástroje ping, odezvu nástrojů pro vzdálenou správu SSH a Telnet, a dále odezvu virtuální stanice při stahování a nahrávání souborů pomocí služby FTP – u vybraných virtualizačních nástrojů.

Jedná se o `CLI` (command line interface, nástroj příkazové řádky) nástroj napsaný v jazyce Python. Nástroj v současné době umožňuje provádět měření všech virtualizačních nástrojů, které podporují správu přes `libvirt` API (application pro-

gramming interface, aplikační rozhraní), např. KVM/QEMU, OpenVZ, LXC apod. a nástroje Docker, spravovaného přes vestavěné REST API. Nástroj `vte` je modulární, takže je možné jej jednoduše rozšířit o podporu dalších služeb, případně virtualizačních nástrojů.

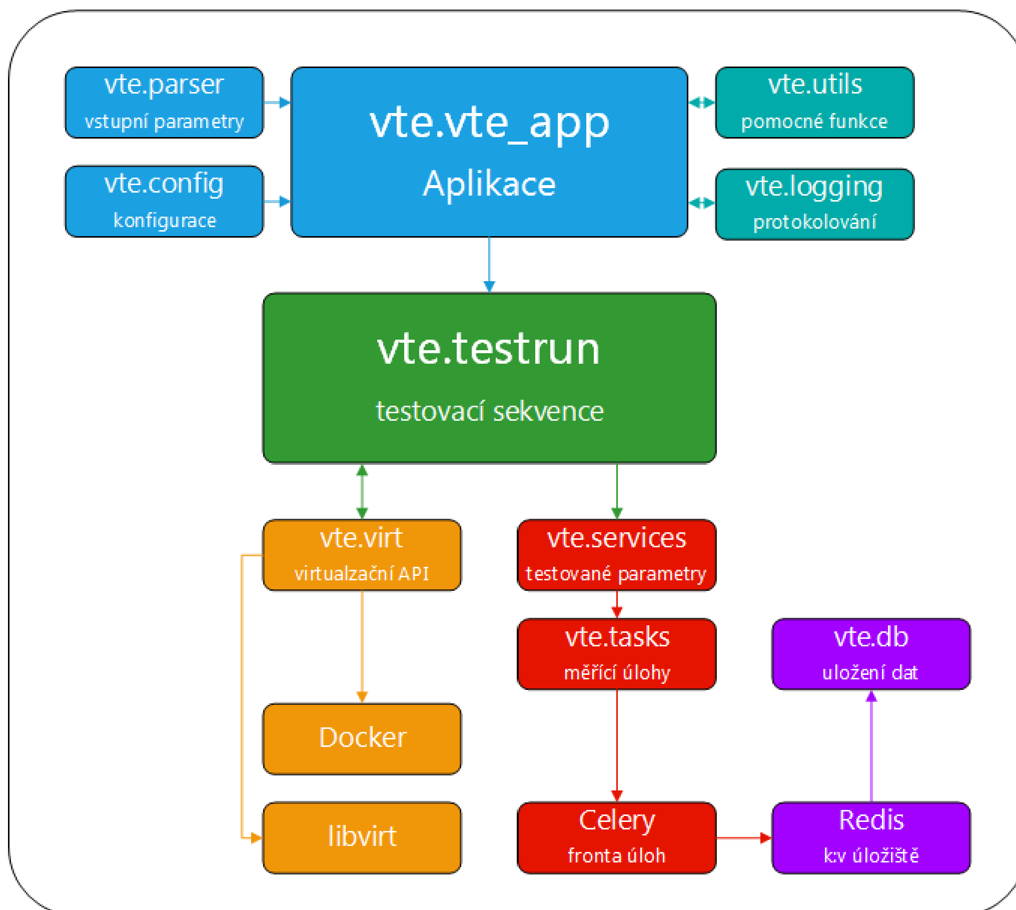
Při samotném měření pomocí nástroje `vte` lze definovat několik parametrů, které ovlivňují způsob měření, např. počet opakování jednotlivých měření, nebo maximální počet virtuálních stanic, který se má otestovat. Výčet všech parametrů nástroje je zobrazen na ukázce 2.24, která obsahuje nápovědu nástroje. Tu lze vyvolat po zadání příkazu `vte --help` do příkazové řádky. Podrobnější rozbor jednotlivých parametrů je potom obsažen v části 2.3.1.

```
$ vte --help
usage: vte [-h] [--version] [-v] [-l LOGFILE] [-r REPEAT] [-n NUM_TESTS]
          [-m MAX_VMS] [-t TAG] [-d DB]
          {ssh,telnet,ftpdown,ftpup,ping}
          [{ssh,telnet,ftpdown,ftpup,ping} ...] {docker,kvm}

positional arguments:
  {ssh,telnet,ftpdown,ftpup,ping}
                        Services to test.
  {docker,kvm}          Virtualization type to use for this run.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -v, --verbose         Console logger verbosity, can be specified multiple
                        times.
  -l LOGFILE, --logfile LOGFILE
                        Optional logfile location. Defaults to /tmp/vte.log
  -r REPEAT, --repeat REPEAT
                        Repeat each test step multiple times.
  -n NUM_TESTS, --num_tests NUM_TESTS
                        If specified, test only <max_tests> VMs for each VM
                        count (default: test all running VMs).
  -m MAX_VMS, --max_vms MAX_VMS
                        Maximum number of VMs/containers to spawn and test.
  -t TAG, --tag TAG     Optional tag to append to all results in this test
                        run.
  -d DB, --db DB        Path to result DB (SQLite3). DB will be created in-
                        memory by default.
```

Ukázka 2.24: Nápověda nástroje `vte`

2.3.1 Popis modulů aplikace `vte`

Aplikace `vte` implementuje kromě většího množství kódu zajišťujícího testovací a měřící logiku také několik různých rozhraní, pomocí kterých komunikuje s různými externími nástroji a aplikacemi, např. s databázovým systémem SQLite3, s virtualizačním nástrojem Docker nebo s asynchronní frontou úloh Celery. Aby byl kód aplikace přehledný a snadno udržovatelný, je rozdělen do bloků, tzv. programových modulů.



Obr. 2.1: Rozdělení aplikace `vte` do modulů

Na obrázku 2.1 je zobrazeno rozdělení aplikace `vte` do jednotlivých modulů i s jejich vzájemnými logickými vazbami. Moduly aplikace jsou popsány v následujících sekcích práce, včetně jejich popisu, funkce, a závislostí na ostatních modulech.

vte.virt

Balík (package) `vte.virt` obsahuje jednotlivé moduly rozhraní, které aplikace `vte` používá pro správu virtualizačních nástrojů. Aplikace `vte` v současnosti podporuje

správu virtualizace `Docker`, a také množství virtualizačních nástrojů, které jsou zahrnuty do `libvirt` API. Aplikaci je případně možné rozšířit o podporu dalších virtualizačních nástrojů přidáním modulu, který bude implementovat rozhraní popsané v sekci 2.3.1.

vte.virt.base_api

Modul `vte.virt.base_api` slouží jako šablona rozhraní pro správu jednotlivých virtualizačních nástrojů, tzn. implementuje návrhový vzor adaptér (*Adapter pattern*). Modul obsahuje deklaraci třídy `BaseAPI`, která implementuje všechny metody potřebné při práci s virtualizačními nástroji. Jedná se o následující metody:

- `get_backend` – Metoda `get_backend` slouží k inicializaci externího API (tj. libvirt nebo Docker REST API). Objekt, který toto API reprezentuje, je následně uložen do atributu třídy `backend`, pomocí kterého se následně k externímu API přistupuje.
- `get_active_vms` – Tato metoda slouží k získání seznamu všech virtuálních stanic, které jsou aktuálně ve spuštěném stavu.
- `get_inactive_vms` – Obdobně jako u předešlé metody, návratovou hodnotou metody `get_inactive_vms` by měl být seznam virtuálních stanic, tento seznam by ale měl obsahovat pouze stanice, které nejsou aktivní.
- `start_vm` – Metoda `start_vm` slouží ke spuštění vybrané virtuální stanice.
- `stop_vm` – Metoda `stop_vm` slouží k zastavení vybrané virtuální stanice.
- `start_all_vms` a `stop_all_vms` – tyto metody slouží ke spuštění, resp. k zastavení všech aktivních nebo naopak neaktivních virtuálních stanic. V aplikaci jsou použity např. před začátkem a po skončení testů.
- `get_vm_ip` – Jedná se o statickou metodu, která slouží k získání IP adresy cílové virtuální stanice.

Definicí této třídy je zajištěno oddělení kódu vyšší úrovně od kódu, který má na starosti správu virtualizačních nástrojů. Funkce vyšších úrovní tedy nemusí pro každý typ virtualizace zajišťovat vlastní logiku, ale vždy volají stejné API. Zmíněná třída `BaseAPI` slouží pouze jako návrhový vzor pro další třídy, které z této třídy dědí a v metodách implementují již specifickou logiku pro daný způsob virtualizace. Aby bylo zajištěno, že podtřídy implementují všechny potřebné funkce pro správu virtualizace, všechny metody třídy `BaseAPI` vyvolávají výjimku `NotImplementedError`, která slouží jako upozornění na to, že se jedná pouze o abstraktní třídu.

vte.virt.docker_api

Závislosti modulu:

- standardní modul `random`,

- modul `vte.config`,
- modul `vte.virt.base_api`,
- externí modul `docker`.

Modul `vte.virt.docker_api` implementuje třídu `DockerAPI`. Tato třída dědí od dříve zmíněné třídy `BaseAPI` a slouží ke správě virtualizačního nástroje Docker. Objekt třídy `DockerAPI` implementuje následující metody:

- `_fltr` – Statická metoda `_fltr` slouží k vytvoření filtru, který při voláních funkcí `DockerAPI` sloužících k získání seznamu kontejnerů zajistí, že seznam kontejnerů bude obsahovat pouze ty kontejnery, které jsou určeny pro měření. Kontejnery jsou rozeznávány na základě značky (`label`) se kterou jsou vytvořeny – kontejnery určené pro měření mají ve svých metadatech položku `service` s hodnotou `vte` (viz ukázka 2.23).
- `get_backend` – Návratovou hodnotou této metody je objekt třídy `docker.Client`, který je následně uložen do atributu třídy `DockerAPI.backend`. Pomocí tohoto objektu následně probíhá komunikace s démonem služby Docker, což umožňuje správu tohoto virtualizačního nástroje. Pro inicializaci objektu `docker.Client` je zapotřebí soket nebo URL, na kterém Docker démon naslouchá. Toto URL je získáno z modulu `vte.config`, který obsahuje většinu konfigurace aplikace `vte`.
- `get_active_vms` – Metoda slouží k získání seznamu všech aktivních kontejnerů, tj. kontejnerů ve stavu `up`. Metoda na pozadí volá metodu `containers` třídy `docker.Client`, která bez dalších parametrů vrací právě seznam aktivních kontejnerů.
- `get_inactive_vms` – Metoda je shodná s předešlou metodou. Na pozadí je také volána metoda `docker.Client.containers`, ale jako parametr je k volání přidán filtr. Metoda poté vrací pouze ty kontejnery, jejichž parametr `status` má hodnotu `created` nebo `exited`, tj. pouze ty kontejnery, které byly korektně vytvořeny a jsou neaktivní.
- `_get_random_container` – Jedná se o interní (*private*) metodu třídy `DockerAPI`. Tato metoda slouží k výběru náhodného kontejneru. Metoda nejprve na základě svého vstupního parametru `active`, který určuje, zda se má vybrat náhodný aktivní nebo neaktivní kontejner, získá seznam kontejnerů, stejně, jako např. metoda `get_active_vms`. Následně je pomocí funkce `random.choice` ze seznamu vybrán náhodný kontejner, který je návratovou hodnotou této metody. V případě, že nejsou dostupné žádné kontejnery, vrací metoda hodnotu `None`.
- `start_vm` – Metoda `start_vm` slouží ke spuštění náhodného kontejneru. Poté, co je tento kontejner vybrán pomocí metody `_get_random_container`, je zavolána metoda třídy `docker.Client.start`, jejímž parametrem je ID daného

kontejneru, který je následně spuštěn.

- `stop_vm` – Metoda `stop_vm` je shodná s předchozí metodou, pouze je následně volána metoda `docker.Client.stop`.
- `stop_all_vms` – Tato metoda nejprve získá seznam všech aktivních kontejnerů a následně je jeden po druhé zastaví pomocí metody třídy `docker.Client.stop`.
- `get_vm_ip` – Jedná se o statickou metodu, jejímž vstupním parametrem je aktivní kontejner. Metoda následně z jeho metadat získá IP adresu, která je pak i její návratovou hodnotou.

vte.virt.libvirt_api

Závislosti modulu:

- standardní moduly `random`, `time`, a `xml.dom`
- moduly aplikace `vte.config`, `vte.logging` a `vte.virt.base_api`
- externí modul `libvirt`

Modul `vte.virt.libvirt_api` implementuje třídu `LibvirtAPI`, která obdobně jako třída `DockerAPI` slouží ke správě virtualizace KVM/QEMU, resp. všech typů virtualizací, které podporují správu pomocí knihovny libvirt. Objekty třídy `LibvirtAPI` implementují následující metody:

- `get_backend` – Tato metoda slouží k inicializaci spojení s knihovnou libvirt. Její návratovou hodnotou je objekt `libvirt.virConnect`, který je následně uložen do atributu `backend` této třídy a zprostředkovává všechny operace s knihovnou libvirt. Metoda volá funkci `libvirt.open`, která jako vstupní parametr přijímá koncový bod libvirt API, ke kterému se má připojit. V případě KVM/QEMU je to URI `qemu:///system`, které je importováno z modulu nastavení aplikace `vte.config`.
- `get_all_vms` – Metoda `get_all_vms` slouží k získání seznamu všech virtuálních stanic, které jsou v systému definované. Tento seznam je návratovou hodnotou funkce `listAllDomains`, kterou implementuje objekt třídy `libvirt.virConnect` a obsahuje jednotlivé objekty třídy `libvirt.virDomain`, která reprezentuje virtuální stanice.
- `get_active_vms` – Tato metoda slouží k získání seznamu aktivních virtuálních stanic. Seznam aktivních stanic je získán ze seznamu všech stanic, nad kterým je pro každý objekt `libvirt.virDomain` v tomto seznamu volána jeho metoda `isActive`.
- `get_inactive_vms` – Metoda sloužící k získání seznamu neaktivních stanic. Metoda je implementována identicky s předešlou, pouze jsou do výsledného seznamu stanic přiřazeny jen ty stanice, u kterých funkce `isActive` nevrátí logickou hodnotu pravdy.

- `_get_random_vm` – Metoda vrací náhodně vybranou virtuální stanici ze seznamu aktivních nebo neaktivních stanic v závislosti na hodnotě vstupního parametru `active` této metody. Pro výběr náhodně stanice je použita funkce `random.choice`.
- `start_vm` – Metoda `start_vm` nejprve vybere náhodnou VM pomocí metody `get_random_vm`, a následně u získaného objektu třídy `libvirt.virDomain` zavolá jeho metodu `create`, která vybranou virtuální stanici spustí. Protože start VM trvá určitý čas, je za start VM zařazen ochranný interval 30 vteřin, aby nedošlo ke stavu, kdy se bude aplikace snažit provést měření na stanici, která ještě není plně spuštěna a tedy neodpovídá.
- `stop_vm` – Metoda funguje na stejném principu, jako metoda `start_vm`, pouze je pro získaný objekt `libvirt.virDomain` volána metoda `shutdown`, která stanici zašle signál `ACPI shutdown`, a tím VM zastaví.
- `stop_all_vms` – Tato metoda nejprve získá seznam všech aktivních VM, a následně pro každou stanici reprezentovanou objektem `libvirt.virDomain` volá jeho metodu `destroy`, která nejprve zašle virtuální stanici stejný signál, jako metoda `shutdown`, a v případě, že na něj stanice do uplynutí časovače nezareaguje, ji ukončí přímo („tvrdé“ vypnutí).
- `get_vm_ip` – Tato metoda je statická a slouží k získání IP síťového rozhraní virtuální stanice. Na rozdíl od Docker kontejnerů nemají VM pod KVM/QEMU tuto informaci ve svých metadatech, proto je nutné použít jiný postup. Metoda nejprve extrahuje informaci o MAC adrese rozhraní virtuální stanice ze souboru XML, který obsahuje parametry této stanice. Tento soubor XML lze získat zavoláním metody `libvirt.virDomain.XMLDesc`. XML soubor je následně rozparsován pomocí funkce `xml.dom.minidom.parseString` a hodnota MAC adresy je z tohoto objektu získána funkcí `getElementsByTagName`. Na základě MAC adresy je poté možné zjistit IP adresu dané VM ze souboru `/var/lib/libvirt/dnsmasq/<název sítě>.leases`, který obsahuje aktuální rezervace, resp. přidělené adresy z vestavěného DHCP serveru.

vte.utils

Závislosti modulu:

- standardní moduly `hashlib`, `time`, `functools`,
- modul aplikace `vte.logging`,
- externí modul `psutil`.

Modul `vte.utils` obsahuje definici několika pomocných funkcí, které nemají návaznost na jiný konkrétní modul. Jsou jimi funkce:

- `timed` – Funkce `timed` implementuje tzv. *dekorátor*, tzn. jedná se o funkci, je-

jíž návratovou hodnotou je jiná funkce. Funkce tedy „obaluje“ jinou funkci, které většinou přidává další funkcionalitu, např. rozšířené protokolování nebo, jako v tomto případě, informaci o době, kterou trvalo vykonání dekorované funkce. Vstupním parametrem dekorátoru `timed` je funkce, která má být dekorována. Tělo funkce `timed` potom obsahuje definici interní funkce, `wrapper`, která nejprve do proměnné `start_time` uloží aktuální čas pomocí funkce `time` ze standardního modulu `time`, poté zavolá dekorovanou funkci, tj. funkci, která byla vstupním parametrem funkce `timed`, a po dokončení této funkce uloží do proměnné `end_time` opět aktuální čas. Interní funkce potom vrací slovníkový objekt (`dictionary`), který obsahuje návratovou hodnotu dekorované funkce a také čas, který zabralo vykonávání této funkce. Tento čas se získá rozdílem proměnných `end_time` a `start_time`. Funkce `wrapper` je navíc dekorována dekorátorem `wraps` ze standardního modulu `functools`, který zajistí, že budou zachována všechna metadata původně dekorované funkce, např. její signatura nebo řetězec s popisem funkce (tzv. `docstring`). Zdrojový kód funkce `timed` si lze prohlédnout na ukázce 2.25

```
1  def timed(function):
2   """A decorator that times function execution."""
3   @wraps(function)
4   def wrapper(*args, **kwargs):
5       start_time = time.time()
6       result = function(*args, **kwargs)
7       end_time = time.time()
8       return {'result': result,
9               'exec_time': end_time - start_time}
10  return wrapper
```

Ukázka 2.25: Kód dekorátoru `timed`

- `check_workers` – Funkce `check_workers` slouží ke kontrole, zda jsou v systému spuštěny procesy `Celery worker`, které vykonávají jednotlivá měření. Tato funkce je volána při startu aplikace `vte`. Pokud není v systému aktivní žádný požadovaný proces, aplikace se ukončí.
  V prvním kroku je získán seznam aktuálně spuštěných procesů pomocí funkce `psutil.process_iter`. Pokud je v názvu procesu řetězec `celery` a proces byl spuštěn s argumentem `worker`, ID tohoto procesu (`PID`) se přidá do seznamu nalezených procesů, který je i návratovou hodnotou této funkce. Pokud v systému žádný proces `Celery worker` spuštěn není, funkce vrátí hodnotu `None`.
- `hashify` – Funkce `hashify` slouží k výpočtu unikátního identifikátoru na základě ID objektu, který je vstupním parametrem této funkce, v operační paměti. Pro výpočet identifikátoru je použita funkce `id`, jejíž výstupem je ce-

ločíselné ID objektu. To je poté převedeno do hexadecimálního formátu, ze kterého je vypočítána MD5 hash. Prvních 10 znaků hashe je poté návratovou hodnotou funkce `hashify`.

vte.services

Závislosti modulu:

- standardní moduly `ftplib`, `io`, `os`, `re`, `subprocess` a `telnetlib`,
- moduly aplikace `vte.config` a `vte.utils`,
- externí moduly `paramiko` a `psutil`.

Modul `vte.services` obsahuje definice funkcí, které realizují měření všech parametrů, které aplikace podporuje. Aktuálně jsou to:

- odezva vzdáleného připojení pomocí SSH,
- odezva vzdáleného připojení pomocí nástroje Telnet,
- přenosové rychlosti, resp. doba stažení a nahrání souboru pomocí protokolu FTP,
- odezva virtuální stanice na `ICMP ECHO` pakety (ping),
- využití procesoru hostitele,
- využití paměti RAM hostitele,
- zátěž hostitele.

V případě, že by bylo požadováno rozšíření aplikace `vte` o měření dalších parametrů virtualizace nebo hostitele, stačí implementovat příslušnou funkci v tomto modulu a následně vytvořit nový objekt `Task`, který bude tuto funkci využívat (bude popsáno v sekci 2.3.1).

Modul `vte.services` obsahuje definici těchto funkcí:

- `check_telnet` – Funkce `check_telnet` slouží k měření odezvy vzdáleného připojení na stanici pomocí protokolu Telnet. Vstupními parametry funkce jsou IP adresa vzdálené stanice a nepovinné parametry `command` (určuje příkaz, který bude po připojení ke vzdálené stanici vykonán) a `credentials`, který umožňuje specifikovat konkrétní sadu přihlašovacích údajů. Pokud není tento parametr specifikován, použijí se výchozí přihlašovací údaje získané z konfigurace aplikace, tj. z modulu `vte.config`.
  Jádrem této funkce je objekt třídy `Telnet` ze standardního modulu `telnetlib`. Po inicializaci tohoto objektu, tj. připojení ke vzdálené stanici, je zavolána jeho metoda `Telnet.read_until`, která vyčítá ze spojení data, až dokud nenarazí na zadaný řetězec, v tomto případě řetězec `login:`, který signalizuje, že vzdálená stanice připojení přijala a očekává zadání přihlašovacích údajů, resp. uživatelského jména. To je zasláno pomocí metody `Telnet.write`. Tento

proces se opakuje znova i pro zadání hesla. Následně jsou z připojení opět vyčítána data do doby, než se vrátí řetězec `]$`, který značí, že přihlášení proběhlo úspěšně a nacházíme se v příkazovém řádku (tzv. `prompt`). Pokud byla funkce volána s parametrem `command`, je následně tento příkaz vzdálené stanici zaslán a je vyčten jeho výstup, který je uložen do proměnné. Připojení je následně ukončeno zasláním řetězce `exit` a zavoláním metody `Telnet.read_all`, která se vrátí až po ukončení spojení.
Funkce `check_telnet` je dekorována dekorátorem `vte.utils.timed`, který poskytuje informace o době vykonávání funkce, což je zároveň i měřená hodnota pro službu Telnet.

- `check_ssh` – Tato funkce slouží k měření odezvy připojení na vzdálenou stanici pomocí SSH. Vstupními parametry funkce jsou obdobně jako u funkce `check_telnet` povinný parametr `host`, určující IP vzdálené stanice, a dále nepovinný parametr `command`, určující případný příkaz, který se má na vzdálené stanici vykonat a parametr `credentials`, pomocí kterého lze specifikovat přístupové údaje. Ty jsou ve výchozím nastavení získány z konfiguračního modulu aplikace `vte.config`.
  Základem funkce `check_ssh` je objekt třídy `SSHClient` z externího modulu `paramiko`. Po inicializaci SSH klienta reprezentovaného tímto objektem je nutné zajistit, aby klient automaticky schvaloval připojení ke stanicím s dosud neznámým otiskem RSA klíče. To zajistí metoda `set_missing_host_key_policy`, jejímž parametrem je politika pro schvalování neznámých otisků RSA klíčů. V tomto případě použijeme politiku `paramiko.AutoAddPolicy`, která automaticky neznámé otisky klíčů schválí.
  Samotné připojení je poté realizováno pomocí metody `SSHClient.connect`, jejímiž vstupními parametry jsou IP stanice `host`, uživatelský účet `user` a heslo nebo cesta k souboru s privátním RSA klíčem `key_filename`. Po připojení je případně vykonán zadaný příkaz pomocí metody `SSHClient.exec_command`, jejímiž návratovými hodnotami je trojice standardních výstupů `stdin`, `stdout` a `stderr`. Výsledek příkazu je poté vyčten z výstupu `stdout` metodou `readlines`.
  Funkce `check_ssh` je dekorována funkcí `vte.utils.timed`, takže její návratovou hodnotou je slovník obsahující výsledek volaného příkazu a doba vykonávání funkce, která je i měřenou hodnotou pro tuto službu.
- `check_ping` – Funkce `check_ping` slouží k měření odezvy virtuální stanice na ICMP pakety. Pro měření odezvy se používá nástroj příkazové řádky `ping`. Příkaz je volán s parametry:
  - `-q` – Tento parametr potlačí většinu výstupu příkazu do konzoly, je zobrazena pouze úvodní zpráva a výsledky.
  - `-c 10` – Pro zpřesnění výsledků měření je zasláno celkem 10 žádostí `ICMP`

`ECHO`.

  - `-i 0.2` – Tento parametr udává interval mezi jednotlivými zaslanými `ICMP ECHO` zprávami, v tomto případě jsou zprávy zasílány každých 200 milisekund pro urychlení měření. Bez zadání tohoto parametru činí interval 1 vteřinu.

  Terminálový příkaz `ping` je z Python interpreteru volán pomocí funkce ze standardního modulu `subprocess.check_output`. Získaný výstup příkazu je poté rozparsován a pomocí funkcí standardního modulu `re` (regulární výrazy) jsou z něj extrahovány požadované hodnoty. Návratovou hodnotou funkce je průměrná doba mezi zasláním `ICPM ECHO` žádosti a obdržení odpovědi (`RTT` – Round-Trip Time).

- `check_ftp_download` – Pomocí této funkce lze změřit čas, za který bude ze vzdáleného FTP souboru stažen cílový soubor. Vstupními parametry funkce jsou povinný parametr `host`, vyjadřující IP adresu FTP serveru a nepovinný parametr `credentials`, který stejně jako u předchozích funkcí určuje přihlašovací údaje k serveru.

  Funkcionalitu zajišťuje objekt třídy `FTP` ze standardního modulu `ftplib`. Po inicializaci objektu, a tedy i spojení s FTP serverem, je nejprve otevřen cílový soubor na lokálním disku, do kterého budou stažená data zapsána. Místo reálného souboru je zde použit objekt `os.devnull`, pomocí kterého lze přistupovat k zařízení `/dev/null`. Stažená data jsou tedy přímo po stažení zahozena. Po připojení na server je nejprve vyhledán cílový soubor s názvem `download.bin` pomocí metody `FTP.nlst`, a dále je do proměnné uložena velikost tohoto souboru (metoda `FTP.size`). Ke stažení souboru potom slouží metoda `FTP.retrbinary`, jejímiž parametry jsou název vzdáleného souboru, a dále objekt, do kterého mají být data zapsána, resp. jeho metoda `write`.

  Po stažení souboru je zkontrolován poslední návratový kód FTP klienta. Kód by měl být roven hodnotě 226, která značí, že datové spojení bylo ukončeno a soubor byl úspěšně přenesen. Návratovou metodou funkce `check_ftp_download` je potom n-tice (`tuple`), která obsahuje návratový kód, název souboru a jeho velikost. Měřenou hodnotou je doba, kterou trvalo vykonání této funkce. Tuto hodnotu zajišťuje stejně jako u předchozích funkcí dekorátor `vte.utils.timed`.

- `check_ftp_upload` – Funkce `check_ftp_upload` zajišťuje měření doby, kterou trvá nahrání souboru na vzdálený FTP server. Funkce má stejné vstupní parametry jako funkce `check_ftp_download`.

  Po inicializaci FTP klienta reprezentovaného objektem `ftplib.FTP` je vygenerován náhodný binární soubor o velikosti 10 MB. Data jsou vygenerována pomocí funkce `os.urandom`, která ze zařízení `/dev/urandom` vyčtou 10 MB náhodných dat. Tato data jsou poté „zabalena“ do objektu `io.BytesIO`, aby bylo

možné je po částech vyčítat metodou `read`, kterou objekt této třídy implementuje. Data jsou pak přenesena na FTP server pomocí metody `FTP.storbinary`. Po ukončení přenosu je již zmíněnou metodou `FTP.nlst` vyhledán nahraný soubor, který je poté metodou `FTP.delete` vymazán. Návratovou hodnotou funkce je potom n-tice obsahující návratový kód klienta po ukončení přenosu, název cílového souboru, jeho velikost a návratový kód operace mazání. Doba vykonávání funkce je opět měřena pomocí dekorátoru `vte.utils.timed`.

- `_odict_to_list` – Jedná se o pomocnou funkci, která slouží k „rozbalení“ objektu třídy `OrderedDict` do seznamu (`List`). K jednotlivým položkám je při rozbalování přidán také prefix, který je vstupním parametrem funkce. Tato funkce slouží ke konverzi hodnot získaných z funkcí pro měření zátěže hostitele do formátu, se kterým následně mohou ostatní funkce jednodušeji pracovat.
- `check_host_cpu` – Funkce `_check_host_cpu` provádí odečet statistik procesoru hostitele. Statistiky jsou získány pomocí funkce `cpu_times_percent` z externího modulu `psutil`, která vrací objekt třídy `OrderedDict`, jehož klíči jsou jednotlivé statistiky procesoru a hodnoty jsou potom procentuální vyjádření procesorového času. Získané hodnoty jsou poté pomocí funkce `_odict_to_list` převedeny do seznamu, a k jednotlivým hodnotám je přidán prefix `host.cpu`.
- `check_host_mem` – Tato funkce je implementována podobně, jako předchozí funkce `check_host_cpu`. Statistiky využití operační paměti hostitele jsou získány z funkce `psutil.virtual_memory`, následně jsou opět převedeny do seznamu pomocí funkce `_odict_to_list` a je k nim přidán prefix `host.mem`.
- `check_host_load` – Poslední z funkcí, které slouží k měření zátěže hostitele je funkce `check_host_load`. Cílem funkce je získat statistiky `load average`, tedy zátěže systému, které reportuje příkaz `uptime`. Tento příkaz je zavolán funkcí `subprocess.check_output` a z jeho výstupu jsou následně jednoduchými operacemi s řetězci extrahovány hodnoty zátěže. K těm je poté přiřazen prefix `host.load`, a výsledný seznam hodnot je touto funkcí navrácen.

Podrobný popis všech hodnot, jejichž měření je realizováno výše popsanými funkcemi, je uveden v kapitole 2.3.2.

vte.tasks

Závislosti modulu:

- moduly aplikace `vte.config` a `vte.services`,
- externí modul `celery`.

Modul `vte.tasks` obsahuje definici úloh pro asynchronní frontu úloh `Celery`, pomocí které jsou realizována všechna měření aplikace `vte`. Kromě definice samotných úloh se v tomto modulu také nachází definice Celery aplikace. Tento modul

je poté importován externími procesy `Celery worker`, které z něj generují seznam dostupných úloh.

Modul obsahuje deklarace následujících tříd:

- `BaseTask` – Třída `BaseTask` obsahuje veškerý společný kód pro všechny ostatní třídy reprezentující jednotlivé typy úloh, které z této třídy dědí. Samotná třída `BaseTask` je potomkem třídy `celery.Task`, která implementuje veškerou logiku vztahující se k frontě úloh `Celery`. Třída `BaseTask` implementuje následující metody:
  - `run` – Všechny třídy, které dědí z třídy `celery.Task`, musejí mít implementovanou metodu `run`. Právě tato metoda je volána procesy `Celery worker`, a vykonává tedy operace dané úlohy. Metoda `run` na pozadí volá metodu `measure` třídy `BaseTask`, která provádí samotné měření. Metoda `run` pomocí bloku k ošetření výjimek (`try ... except`) zajišťuje, že v případě, že metoda `measure` neočekávaně skončí předem definovanou výjimkou (např. výjimkou `ConnectionRefusedError`, která je vyvolána v případě nezdařeného spojení), bude selhané měření opakováno. Fronta úloh `Celery` se pokusí o opakování úlohy po uplynutí ochranného intervalu `TASK_COUNTDOWN`, který je definovaný v nastavení aplikace `vte`. Maximální počet opakování selhaných úloh určuje parametr `TASK_RETRY_COUNT`, který je rovněž definován v nastavení aplikace. Metoda `run` dále obsahuje kód sloužící k protokolování diagnostických informací o úlohách a její návratovou hodnotou je hodnota metody `measure`.
  - `measure` – Metoda `measure` je ve třídě `BaseTask` pouze abstraktní. Třídy, které ze třídy `BaseTask` dědí, musí vždy deklarovat vlastní metodu `measure`. V opačném případě je při zavolání této metody vyvolána výjimka typu `NotImplementedError`.
- `SSHTask` – Tato třída reprezentuje úlohu měření odezvy SSH. Třída definuje pouze metodu `measure`, která volá funkci `check_ssh` z modulu `vte.services`.
- `TelnetTask` – Stejně jako předchozí třída představuje tato třída jednu z měřících úloh, konkrétně měření odezvy vzdáleného připojení protokolem Telnet. Metoda `measure` této úlohy volá funkci `vte.services.check_telnet`.
- `FTPDownTask` – Třída představuje úlohu měření stažení souboru z FTP serveru. Její metoda `measure` volá funkci `check_ftp_download`.
- `FTPUpTask` – Třída reprezentuje měření nahrání souboru na FTP server. Metoda `measure` této třídy volá na pozadí funkci `check_ftp_upload` z modulu `vte.services`.
- `PingTask` – Poslední ze tříd reprezentujících měření je třída `PingTask`. Třída představuje úlohu měření odezvy stanice na `ICMP ECHO` pakety a na pozadí volá funkci `vte.services.check_ping`.

vte.parser

Závislosti modulu:

- standardní modul `argparse`,
- moduly aplikace `vte` a `vte.config`.

Modul `vte.parser` obsahuje definici uživatelského rozhraní aplikace `vte`. Modul obsahuje deklaraci funkce `setup_parser`, která zajišťuje definici rozhraní včetně všech podporovaných parametrů a příkazů. Pro extrakci argumentů z příkazové řádky je použit objekt `ArgumentParser` ze standardní knihovny `argparse`, který zajišťuje veškerou logiku. Pomocí jeho metod `add_argument` jsou k němu potom přiřazeny všechny podporované příkazy a parametry. Ukázka rozhraní aplikace, konkrétně nápověda k příkazu, byla již uvedena na ukázce 2.24. Rozhraní podporuje následující parametry a příkazy:

- `--version` – při spuštění aplikace s tímto parametrem dojde k vypsání verze aplikace do konzoly, aplikace je následně ukončena.
- `-v`, `--verbose` – Tento parametr určuje úroveň protokolování, resp. úroveň diagnostických zpráv, které budou při běhu aplikace vypisovány do konzoly. Při výchozím nastavení aplikace do konzoly vypisuje pouze zprávy úrovně `ERROR` a závažnějších. Tento parametr je možné specifikovat víckrát, a zobrazit tak i zprávy úrovní `INFO` a `DEBUG`.
- `-l`, `--logfile` – Parametr `logfile` určuje umístění protokolovacího souboru. Pokud není zadán, je protokolovací soubor zapsán do umístění definovaném v modulu nastavení aplikace, `vte.config`.
- `-r`, `--repeat` – Tento parametr umožňuje určit počet opakování testovacích kroků během testování jednoho počtu aktivních VM, např. při hodnotě parametru 3 by testy všech služeb byly pro každý počet aktivních VM provedeny celkem třikrát. Výchozí hodnotou parametru je 1.
- `-n`, `--num_tests` – Tento parametr udává počet stanic, na kterých budou provedeny během jednoho testovacího kroku testy zadaných služeb. Pokud bude např. hodnota parametru 10, testy služeb budou provedeny na 10 náhodně vybraných VM. Pokud není hodnota zadána, testy proběhnou vždy na všech aktivních VM v daném kroku.
- `-m`, `--max_vms` – Tento parametr umožňuje zadat maximální počet virtuálních stanic, pro který mají být provedeny testy. Výchozí hodnotou je 10 stanic. Pokud je zadaná hodnota větší, než počet VM dostupných v systému, test je ukončen po otestování počtu všech dostupných stanic.
- `-t`, `--tag` – Parametrem `tag` je určena značka, která bude přidána ke každé naměřené hodnotě během ukládání dat do databáze.
- `-d`, `--db` – Parametr `db` určuje cestu k souboru databáze SQLite3, do kterého

budou zapisována naměřená data. Pokud není zadán, bude databáze vytvořena pouze v paměti RAM, takže naměřená data nebudou uchována.
- `services` – Jedná se o povinný parametr, který určuje, které služby budou otestovány. Vždy je nutné zadat alespoň jednu službu. Možné hodnoty jsou `ssh`, `telnet`, `ping`, `ftpup` a `ftpdown`.
- `virt_type` – Tento parametr určuje, jaký virtualizační nástroj má být otestován. Je nutné zadat právě jednu hodnotu. Možné hodnoty jsou `docker` nebo `kvm`.

vte.logging

Závislosti modulu:
- standardní modul `inspect`,
- standardní modul `logging`,
- standardní modul `os`.

Modul `vte.logging` zajišťuje protokolování pro zbytek aplikace `vte`. Modul obsahuje deklaraci následujících funkcí a tříd:
- `setup_logging` – Jedná se o funkci, která zajišťuje prvotní nastavení protokolování při startu aplikace. Nejprve je definován protokolovací objekt (`logger`) nejvyšší úrovně s názvem `vte`. Nastavení protokolování se potom provádí úpravou atributů tohoto objektu. Konkrétně je nastavena úroveň protokolování na základě vstupních parametrů aplikace (parametr `verbose`) pomocí metody `logger.setLevel` a k protokolovacímu objektu jsou následně připojeny dva další objekty, které zajišťují výstup – jedná se o tzv. *handlery.* Protokolovací záznamy se jednak ukládají do definovaného souboru pomocí objektu `logging.FileHandler` a výstup je také přesměrován do konzole pomocí objektu `logging.StreamHandler`. Protokolovací objekt existuje v globálním prostoru aplikace, takže po jeho nastavení lze následně z libovolného modulu aplikace pomocí funkce `logging.getLogger` tento objekt uložit do libovolné proměnné a následně vytvářet protokolovací zprávy pomocí jeho metod, např. obecné zprávy metodou `logger.log`, zprávy s úrovní `WARNING` pomocí metody `logger.warning` apod.
- `ExtLog` – Třída `ExtLog` obsahuje deklaraci objektu, který umožňuje přidávat do protokolovacích zpráv dodatečné informace, konkrétně do zpráv zahrnuje název funkce, která protokolovací zprávu vyvolala, včetně čísla řádku zdrojového kódu a názvu souboru/modulu, ve kterém je funkce definována.

  Objekty třídy `ExtLog` implementují metodu `log`, která slouží k vygenerování protokolovací zprávy se zmiňovanými informacemi, a dále metody `critical`, `error`, `warning`, `info` a `debug`, pomocí kterých lze vytvořit záznam s patřičnou

„závažností“ (tzv. `severity`).

vte.db

Závislosti modulu:

- standardní modul `sqlite3`,
- modul aplikace `vte.logging`.

V rámci modulu `vte.db` je deklarována třída `ResultDB`, která zastřešuje logiku ukládání výsledků jednotlivých testů do databáze. Pro uložení je použita databáze `SQLite3`, se kterou je možné pracovat přímo na disku nebo v paměti. Databáze `SQLite3` sice na rozdíl od „plnohodnotných“ databází `MySQL` nebo `PostgreSQL` disponuje menším množstvím funkcionalit, ale díky tomu je jednoduchá, nenáročná a nezávislá na externích knihovnách nebo balíčcích.

Objekty třídy `ResultDB` implementují následující metody:

- `_prepare_db` – Jedná se o interní metodu, která slouží k přípravě databáze (vytvoření struktur potřebných pro ukládání dat), pokud se jedná o novou databázi. Pomocí SQL dotazu `CREATE TABLE IF NOT EXISTS results` je vytvořena tabulka `results` s následující strukturou:
  - `timestamp` (text) – pole sloužící pro uložení unixové časové značky pro daný záznam,
  - `test_run_id` (text) – ID testovací sekvence (podrobně popsáno v sekci 2.3.1),
  - `vm_count` (celé číslo) – počet VM, které byly aktivní v době vytvoření záznamu,
  - `metric` (text) – typ záznamu / měřeného parametru,
  - `value` (desetinné číslo) – hodnota měřeného parametru,
  - `tag` (text) – volitelná značka, na základě které lze rozlišit hodnoty z různých testovacích sekvencí. Seznam použitých značek je uveden v kapitole 3.1.
- `_query` – Tato metoda slouží pro vykonání SQL dotazu nad databází a následnému ukončení dané databázové transakce a je volána ostatními metodami třídy `ResultDB`.
- `store_result` – Metoda slouží k uložení naměřených výsledků do databáze, kdy s poskytnutými hodnotami volá interní metodu `_query`, která následně SQL dotazem `INSERT INTO` vloží data do vytvořené tabulky `results`.

vte.config

Závislosti modulu:

- standardní modul `os`.

Modul `vte.config` obsahuje definici různých nastavení pro zbytek aplikace `vte`, mezi které patří:

- přihlašovací údaje pro služby Telnet, SSH a FTP,
- adresa URL rozhraní API nástroje Docker,
- adresa URL rozhraní API nástroje KVM/libvirt,
- parametry sítě nástroje KVM/libvirt (cesta k souboru s přidělenými IP adresami z DHCP serveru, prefix MAC adres jednotlivých VM, IP podsíť síťových rozhraní VM),
- cesta k souboru protokolu,
- parametry pro úlohy definované v modulu `vte.tasks`, jmenovitě počet opakování selhaných úloh a ochranný interval mezi selhanými úlohami.

Pro každý zmíněný parametr je v souboru definovaná výchozí hodnota. V případě, že by ale bylo nutné tyto hodnoty změnit (typicky např. přihlašovací údaje k jednotlivým testovaným službám), by bylo nutné toto nastavení v souboru s tímto modulem přepsat. Protože by byl tento přístup značně nepohodlný, je možné všechny zmíněné parametry definovat pomocí proměnných prostředí (tzv. *Environment Variables*, nebo `ENVVARS`). V případě, že je parametr pomocí této proměnné definován, použije se hodnota definovaná v prostředí, v opačném případě platí výchozí hodnota definovaná přímo v tomto modulu. Seznam všech výchozích hodnot aplikace definovaných v tomto modulu je obsažen v příloze B.3.

vte.testrun

Závislosti modulu:

- standardní moduly `random` a `time`,
- moduly aplikace `vte.db`, `vte.logging`, `vte.services`, `vte.tasks`, `vte.utils`,
- externí modul `celery`.

Modul `vte.testrun` obsahuje deklaraci stejnojmenné třídy `TestRun`. Objekt této třídy implementuje veškerou logiku spojenou se spouštěním a vykonáváním testů nebo jejich jednotlivých kroků. Objekt této třídy je při inicializaci vytvořen s atributy, které následně určují jednotlivé parametry testů, jako je typ virtualizace, výčet měřených služeb, maximální počet spuštěných VM nebo počet opakování jednotlivých testů a testovacích kroků.

Objekty třídy `TestRun` implementují následující metody:

- `_generate_tasks` – Vstupním parametrem této metody je seznam virtuálních stanic a typ služeb, které mají být otestovány. V rámci metody jsou poté vygenerovány objekty představující konkrétní měřící úlohy pro vybrané VM, které jsou následně odeslány do fronty úloh ke zpracování.

- `check_host` – Metoda `check_host` slouží ke změření zátěže a využití zdrojů hostitele. Metoda změří tyto hodnoty pomocí funkcí uvedených v sekci 2.3.1 a následně je uloží do databáze výsledků.
- `test_step` – Tato metoda představuje jeden testovací krok, tzn. změření parametrů vybraných služeb spuštěných na jednotlivých virtuálních stanicích.
  V rámci metody je nejprve získán seznam všech aktuálně aktivních stanic. Z těchto stanic je následně vybrán fixní počet, který má být otestován v závislosti na parametru `max_tests` pro danou testovací sekvenci. Ze seznamu aktivních stanic jsou opakovaně náhodně vybírány stanice do té doby, než je připraven daný počet stanic. Aplikace také umožňuje v každém kroku změřit parametry vždy pro všechny aktivní stanice.
  Po vybrání VM, jejichž parametry budou změřeny, je následně vygenerován seznam měřících úloh pomocí výše popsané metody `_generate_tasks`. Úlohy jsou vygenerovány vždy jen pro jednu službu a testy dalších služeb jsou zahájeny vždy až po jejich dokončení. Vygenerované úlohy jsou následně zaslány ke zpracování do fronty úloh `Celery`, kde je vykoná speciální proces (`Celery worker`) spuštěný paralelně s aplikací. Úlohy je možné zpracovávat sekvenčně, případně paralelně ve větším množství zároveň – to lze určit při spuštění `Celery worker` procesu jeho parametrem `concurrency`.
  Po zaslání úloh do asynchronní fronty aplikace čeká na jejich dokončení, resp. v pravidelných intervalech testuje, zda jsou již výsledky všech úloh připraveny. Pokud ne, aplikace se na krátkou dobu uspí a následně provede kontrolu znovu. Pokud jsou výsledky již připraveny, aplikace je uloží do databáze výsledků a celý proces se opakuje i pro další služby v daném testovacím kroku.
- `test_vm_count` – Metoda `test_vm_count` slouží k provedení testu pro konkrétní počet aktivních VM, který je daný jejím vstupním parametrem `count`. Před zahájením testů jsou nejprve spuštěny další VM tak, aby bylo dosaženo jejich požadovaného počtu. O samotné testy se pak stará metoda `test_step` popsaná výše. Pro daný počet VM je možné testovací kroky několikrát opakovat. Počet opakování je určen parametrem `repeat_tests`, který opět platí pro celou testovací sekvenci (`TestRun`).
- `run` – Pomocí metody `run` je celá testovací sekvence spuštěna. Před zahájením a po skončení testů metoda zajistí, že jsou všechny VM zastaveny, a pak pro jednotlivé počty aktivních VM v intervalu 0 až `max_vms` volá opakovaně metodu `test_vm_count` s požadovanou hodnotou.

vte.vte_app

Závislosti modulu:

- standardní modul `sys`,
- moduly aplikace `vte.logging`, `vte.parser`, `vte.testrun`, `vte.utils`, `vte.virt`

Modul `vte_app` je hlavním „vstupním bodem“ (*entrypoint*) aplikace `vte`. Obsahuje pouze definici funkce `main`, která zajišťuje spuštění aplikace, resp. přípravu jejich jednotlivých komponentů v tomto pořadí:

- získání vstupních parametrů aplikace (funkce `vte.parser.setup_parser`),
- spuštění protokolovacích komponentů (funkce `vte.logging.setup_logging`),
- ověření, že je spuštěn alespoň jeden `celery worker` proces (pomocná funkce `vte.utils.check_workers`),
- vytvoření virtualizačního rozhraní (inicializace objektu `vte.virt.DockerAPI` nebo `vte.virt.LibvirtAPI`),
- inicializace a spuštění testovací sekvence (objekt `vte.testrun.TestRun`).

Ostatní moduly

Kromě hlavních modulů zmíněných v minulých sekcích je v aplikaci `vte` definováno ještě několik pomocných modulů a souborů, které mají různé funkce. Těmito moduly jsou:

- modul `vte.__init__` – Tento speciální modul obsahuje definici metadat o aplikaci (autor, verze, atp.) a jeho přítomnost v adresáři `vte` také zajišťuje, že je adresář zároveň balíčkem (*package*), který lze případně importovat do jiných modulů.
- modul `vte.__main__` – Přítomnost tohoto modulu v hlavním adresáři aplikace zajišťuje, že aplikaci lze spustit např. ze souboru ZIP jeho zavoláním ve formě příkazu `python <soubor>.zip` z příkazové řádky.
- soubor `vte_run.py` – Tento soubor se nachází mimo definici balíčku `vte` a slouží jako vstupní bod aplikace při volání např. z příkazové řádky.
- soubor `setup.py` – Jedná se o řídící soubor pro balíček `setuptools`, který slouží pro instalaci aplikace. V rámci tohoto souboru jsou definována všechna potřebná metadata, závislosti aplikace a také její vstupní body, takže lze celou aplikaci jednoduše nainstalovat jedním příkazem `python setup.py install`. Balíček `setuptools` následně zajistí stažení a instalaci všech závislostí aplikace a vytvoření patřičných odkazů na aplikaci v adresářích se spustitelnými soubory (typicky např. `/bin`, `/usr/bin`, případně `/usr/local/bin`), aby bylo možné aplikaci spouštět zadáním jejího názvu do příkazové řádky.

2.3.2 Výstupní data a popis naměřených hodnot

V kapitolách 2.3.1 a 2.3.1 bylo popsáno, jakým způsobem jsou měření jednotlivých služeb prováděna a jak jsou následně získaná data uložena pro další zpracování.

Jednotlivé parametry služeb jsou nazývány jako metriky, přičemž jsou v databázi navzájem rozlišeny podle stejnojmenného sloupce. Tato kapitola obsahuje výčet jednotlivých metrik včetně jejich popisu.

Systémové metriky – CPU

Metriky vyjadřující zatížení a využití CPU obsahují v názvu prefix `host.cpu`. Mezi tyto metriky patří:

- `host.cpu.user` – procentuální vyjádření celkového procesorového času, který byl využit pro obsluhu uživatelských procesů (aplikací).
- `host.cpu.nice` – metrika je shodná s předchozí, ale jsou do ní zahrnuty pouze procesy s definovanou prioritou (tzv. `nice` parametrem).
- `host.cpu.sys` – tato metrika vyjadřuje procento procesorového času, které bylo využito pro obsluhu systémových procesů (jádra).
- `host.cpu.iowait` – Procentuální vyjádření času, kdy byly procesory nečinné z důvodu čekání na dokončení V/V (vstupně-výstupní) operace.
- `host.cpu.irq` a `host.cpu.soft` – Tyto metriky vyjadřují procentuální podíl procesorového času, který byl využit pro obsluhu hardwarových a softwarových přerušení (*Interrupts*).
- `host.cpu.steal` – Tato metrika se vztahuje pouze k virtuálním systémům a vyjadřuje čas, po který virtuální procesor čekal na obsluhu, zatímco fyzický procesor obsluhoval jiný virtuální procesor.
- `host.cpu.guest` – Procentuální vyjádření procesorového času hostitele stráveného obsluhou virtuálních procesorů.
- `host.cpu.idle` – vyjadřuje procento času, během kterého byly procesory v nečinnosti a nebyl žádný nevyřízený požadavek na V/V operace.

[23]

Systémové metriky – paměť

Následující metriky vyjadřují využití operační paměti RAM. Pokud není uvedeno jinak, všechny metriky vyjadřují hodnotu využité paměti v bajtech.

- `host.mem.total` – celková velikost operační paměti systému.
- `host.mem.available` – velikost paměti, která může být okamžitě přidělena procesu žádajícímu o další alokaci. Na OS Linux je tato metrika dána součtem metrik `free`, `buffers`, a `cached`, viz dále.
- `host.mem.used` – orientační hodnota využité paměti.
- `host.mem.free` – velikost paměti, která není využívána (tzn. obsahuje pouze nulové bity).
- `host.mem.active` – velikost paměti, která je aktuálně využívána procesy.

- `host.mem.inactive` – velikost paměti, která je systémem označena jako nevyužívaná.
- `host.mem.buffers` – velikost paměti alokovaná jako vyrovnávací paměť, např. pro metadata souborového systému.
- `host.mem.cached` – velikost paměti využitá jako vyrovnávací paměť pro různé jiné účely.
- `host.mem.percent` – procentuální využití paměti, které je získáno výpočtem z metrik `total` a `available`.

[24]

Systémové metriky – provozní zátěž

Následující metriky vyjadřují odhad provozního zatížení systému, který je získán přímo ze systému pomocí příkazu `uptime` nebo přečtením souboru `/proc/loadavg`. Hodnota vyjadřuje průměrný počet procesů, které aktuálně procesor využívají, čekají na přidělení procesorového času, případně čekají na realizaci V/V operace. Hodnota je průměrována pro intervaly 1, 5 a 15 min, čemuž odpovídají i názvy metrik:

- `host.load.1min`
- `host.load.5min`
- `host.load.15min`

Průměrná zátěž není na systémech s více CPU žádným způsobem normalizována, při interpretaci těchto hodnot je tedy potřeba brát v úvahu i počet logických jader CPU. Např. hodnota 1,00 na systému s jedním jádrem znamená, že je systém plně vytížen po celou dobu intervalu, naopak identická hodnota na systému se čtyřmi jádry znamená, že procesor byl na měřeném intervalu ze 75% nečinný.[25]

Metriky služeb

Aplikace `vte` naměřené hodnoty parametrů jednotlivých testovaných služeb ukládá do databáze jako tyto metriky:

- `virt.ssh.single` – Tato metrika vyjadřuje odezvu nástroje pro vzdálené připojení SSH. Hodnota vyjadřuje čas v sekundách, který trvají následující kroky:
  1. otevření SSH spojení na vzdálený server (VM),
  2. zavolání příkazu `uptime` na vzdáleném serveru,
  3. ukončení SSH spojení.
- `virt.telnet.single` – Tato metrika je identická s předchozí, pouze platí pro připojení na vzdálený server pomocí služby Telnet.
- `virt.ping.single` – Metrika `ping` vyjadřuje odezvu síťového rozhraní vzdálené stanice. Hodnota udává čas, který uplynul od zaslání 10 ICMP ECHO žádostí na vzdálený server v intervalu 200 ms a obdržení všech ICMP ECHO odpovědí.

- `virt.ftpdown.single` – Hodnota této metriky udává čas v sekundách, který trvá vykonání následujících kroků:
    1. otevření FTP spojení na vzdálený server (VM),
    2. získání seznamu souborů na serveru a nalezení cílového souboru,
    3. stažení souboru o velikosti 10MB a jeho „zapsání“ do `/dev/null`,
    4. ukončení FTP spojení.
- `virt.ftpup.single` – Podobně jako u předchozí metriky, i zde naměřená hodnota vyjadřuje čas v sekundách. Měří se doba provedení následujících kroků:
    1. otevření FTP spojení na vzdálený server (VM),
    2. nahrání souboru o velikosti 10MB na server,
    3. vypsání seznamu souboru na serveru a vyhledání nahraného souboru,
    4. smazání nahraného souboru.

3 VÝSLEDKY MĚŘENÍ

Tato kapitola obsahuje rozbor všech naměřených hodnot a statistik, které byly získány pomocí nástroje `vte`, který byl podrobně rozebrán v předchozí kapitole. Kapitola je rozdělena do dvou částí, z nichž první (3.1) se věnuje porovnání obou virtualizačních nástrojů, jejichž charakteristiky byly změřeny, tj. Docker a KVM. Parametry a charakteristiky všech služeb byly porovnávány při různých úrovních zatížení systému, na kterém byly VM spuštěny.

Druhá část kapitoly (3.2) obsahuje porovnání a výsledky měření parametrů služeb pro obě použité virtualizace, kdy byly postupně a izolovaně zatěžovány jednotlivé součásti systému – CPU, operační paměť, a disky.

Během jednotlivých měření byly jednotlivé komponenty systému hostitele postupně uměle zatěžovány v několika různých úrovních pomocí aplikace `stress`, jejíž parametry jsou krátce popsány v tabulce 3.1.

Tab. 3.1: Parametry aplikace `stress` použité při měření

Tag	Parametry	Tag	Parametry	Tag	Parametry
stress-0	-	mem-8	`-m 8 --vm-bytes 2G`	cpu-8	`-c 8`
stress-8	`-c 2 -i 2 -m 2 -d 2`	mem-12	`-m 12 --vm-bytes 2G`	cpu-12	`-c 12`
stress-20	`-c 5 -i 5 -m 5 -d 5`	mem-16	`-m 16 --vm-bytes 2G`	cpu-16	`-c 16`
io-4	`-d 2 --hdd-bytes 10G -i 2`	mem-20	`-m 20 --vm-bytes 2G`	cpu-20	`-c 20`

V tabulce 3.1 je uvedeno mapování značek (tzv. *tagů*), kterými jsou označena jednotlivá měření, na parametry aplikace stress, se kterými byla aplikace při daném měření spuštěna. Jednotlivé parametry mají následující význam:

- `-c` – Parametr `-c` nebo `-cpu` udává počet vláken aplikace, která budou spuštěna. Tato vlákna dokola volají funkci `sqrt()` nad velkými čísly, čímž dokážou plně zatížit jedno jádro procesoru.
- `-m` – Tento parametr udává počet vláken, která budou po spuštění dokola alokovat a uvolňovat operační paměť funkcemi `malloc()` a `free()`.
- `--vm-bytes` – Tento parametr je doplňkový k parametru `-m` a udává objem paměti, který má jedno vlákno alokovat. Výchozí hodnota je 256 MB.
- `-d` – Parametr `-d` určuje počet vláken, která zatěžují pevné disky opakovaným čtením a zápisem.
- `--hdd-bytes` – Tento doplňkový parametr k parametru `-d` udává max. velikost jednoho souboru na disku, výchozí hodnota je 1 GB.
- `-i` – Tento parametr udává počet procesů, které opakovaně vyprazdňují V/V vyrovnávací paměť na disk.

Měření s výše uvedenými parametry byla opakována pro obě virtualizační technologie, tj. KVM i Docker. V grafech, které se nachází v dalších částech kapitoly

jsou jednotlivé závislosti označeny právě těmito tagy, kde prefix tagu (`docker-` nebo `kvm-`) značí použitou technologii při daném měření.

Všechna měření byla provedena na serveru Cisco UCS B200 M2 s následujícími parametry:

- CPU: 2x Intel Xeon X5680 (12 vláken, 3,33 GHz),
- RAM: 12x 4 GB DDR3-1333 MHz,
- Disky: 12x Seagate Cheetah 300 GB SAS, 15000 RPM, konfigurace RAID 1+0.

3.1 Srovnání nástrojů Docker a KVM

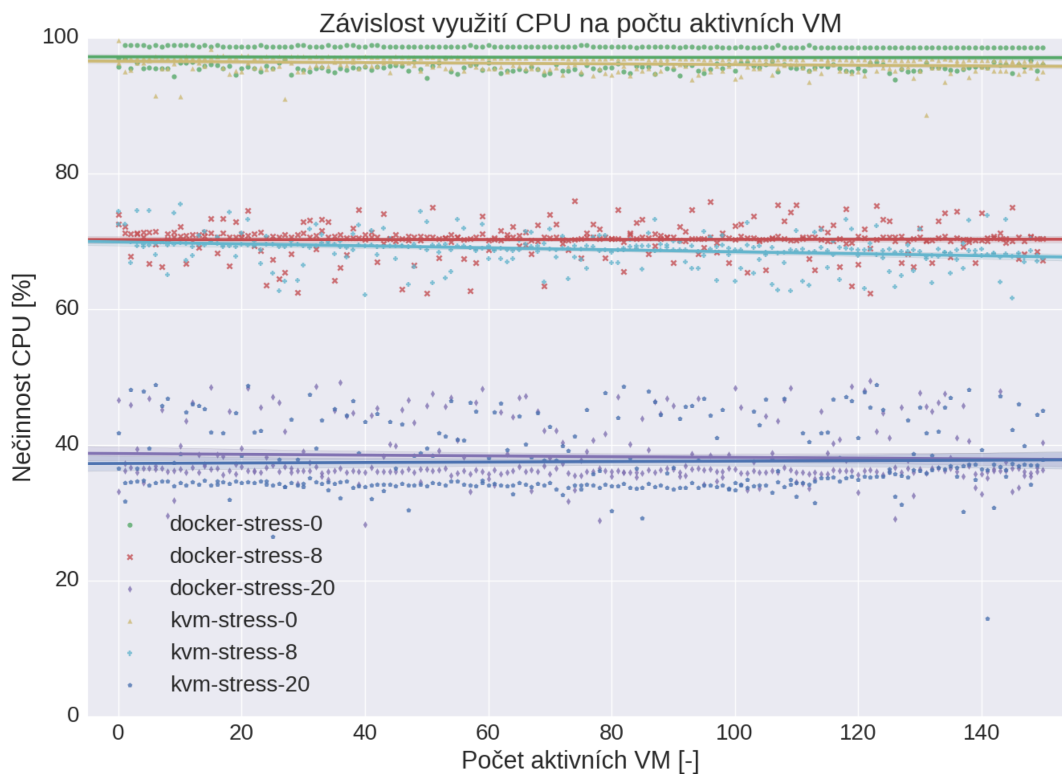
Tato část práce se věnuje srovnání obou technologií, resp. hodnot, které byly naměřeny při jejich použití. Celkem byla pro každou virtualizační technologii provedena 3 měření, jedno na systému bez zátěže (tag `stress-0`), jedno při cca 30% zatížení (tag `stress-8`) a jedno při zatížení na přibližně 80% (tag `stress-20`). Při použití virtualizační technologie Docker bylo úspěšně provedeno měření i při zatížení systému na 100%, u KVM měření nebylo možné provést, protože start první VM trval příliš dlouhou dobu, a nástroje knihovny libvirt v důsledku toho vždy vyvolaly výjimku překročení časového limitu pro start VM. Měření u virtualizace Docker bylo tedy provedeno znovu pro zatížení 80%, kdy bylo test možné provést i pro KVM.

3.1.1 Zátěž systému

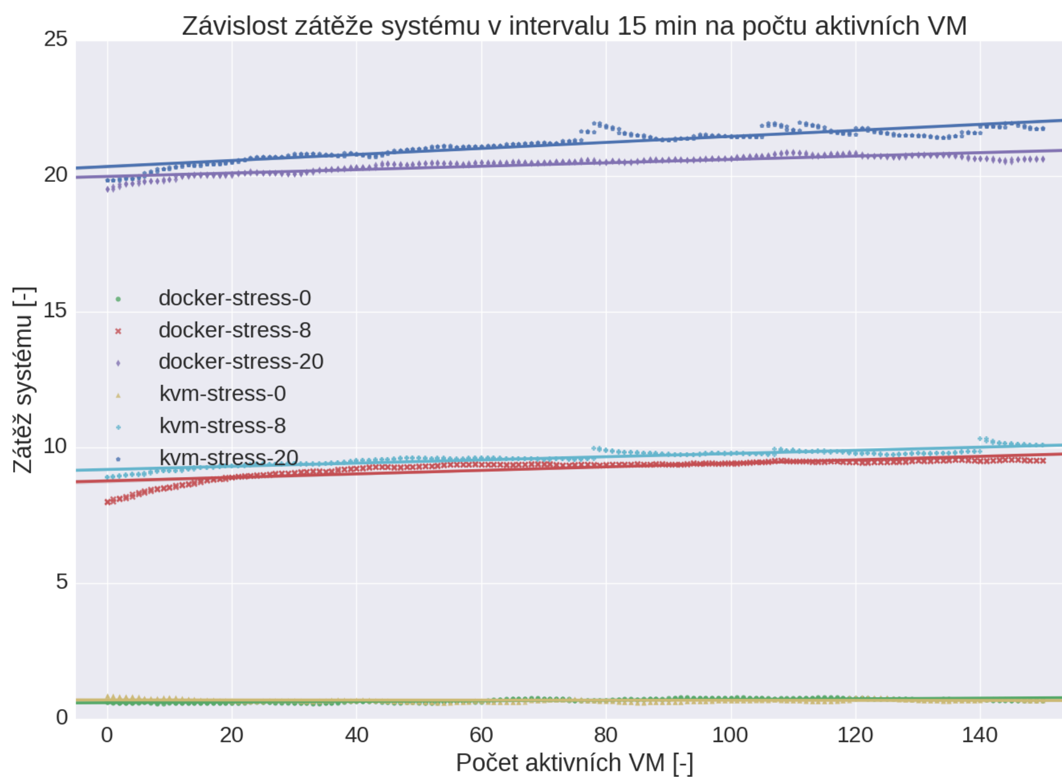
Na obrázku 3.1 je zobrazena závislost využití CPU na počtu aktivních VM při jednotlivých měřeních. Ze zobrazených závislostí můžeme vidět, že zátěž CPU zůstává s počtem aktivních VM prakticky konstantní a je pro obě technologie srovnatelná – v případě Dockeru jsou dosahované hodnoty využití CPU při všech úrovních zátěže přibližně o procento nižší, tato hodnota je ale zanedbatelná. Hodnoty jsou pro zvětšující se počet aktivních VM konstantní z toho důvodu, že jsou procesy spuštěné uvnitř VM po jejich startu nečinné, resp. je v nich spuštěno pouze minimum procesů potřebných pro měření, které vykonávají činnost pouze v případě, že aplikace `vte` provádí oproti dané VM měření.

Podobný trend je možné pozorovat také u obrázku 3.2, kde je zobrazena závislost metriky `load` v intervalu 15 min na počtu aktivních VM. Protože metrika `load` vyjadřuje počet procesů čekajících na nebo obsluhovaných procesorem, odpovídají trendy v grafu očekávaným závěrům (čím více spuštěných VM, tím více procesů, a tím i větší hodnota `load`). Hodnoty pro technologie Docker a KVM jsou opět srovnatelné.

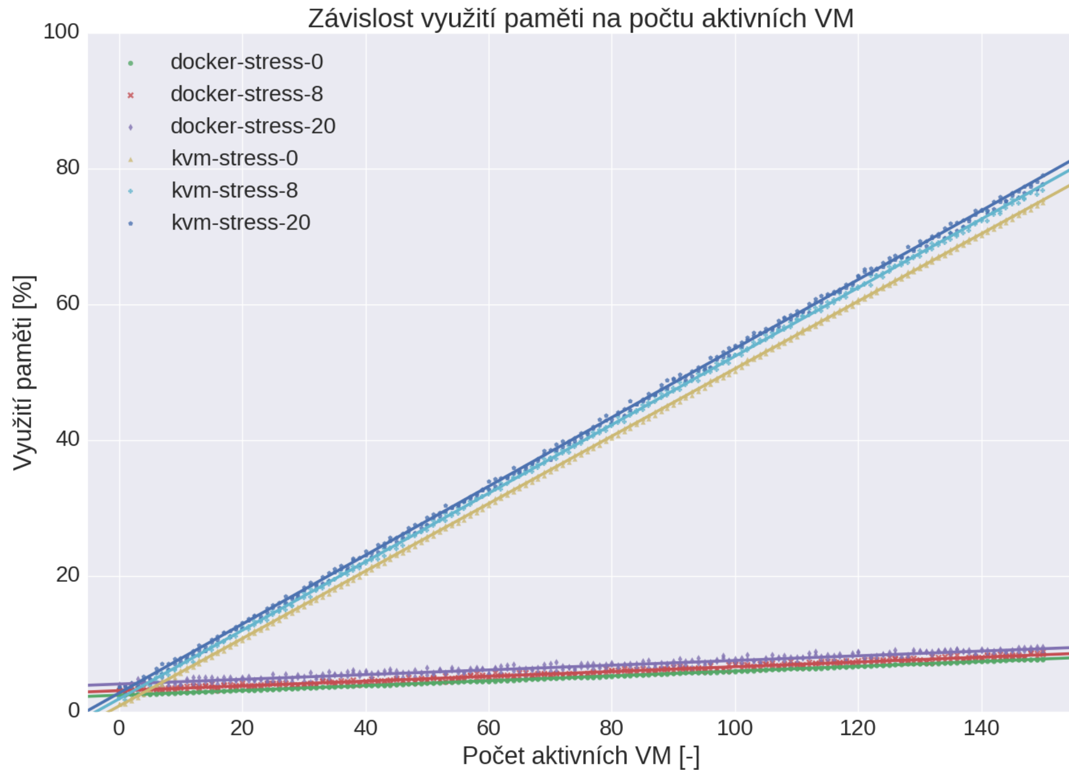
Na posledním obrázku týkajícím se využití systémových zdrojů, 3.3, je zobrazeno využití operační paměti v systému. U Docker kontejnerů činí při jejich maximálním



Obr. 3.1: Graf využití CPU při jednotlivých měřeních



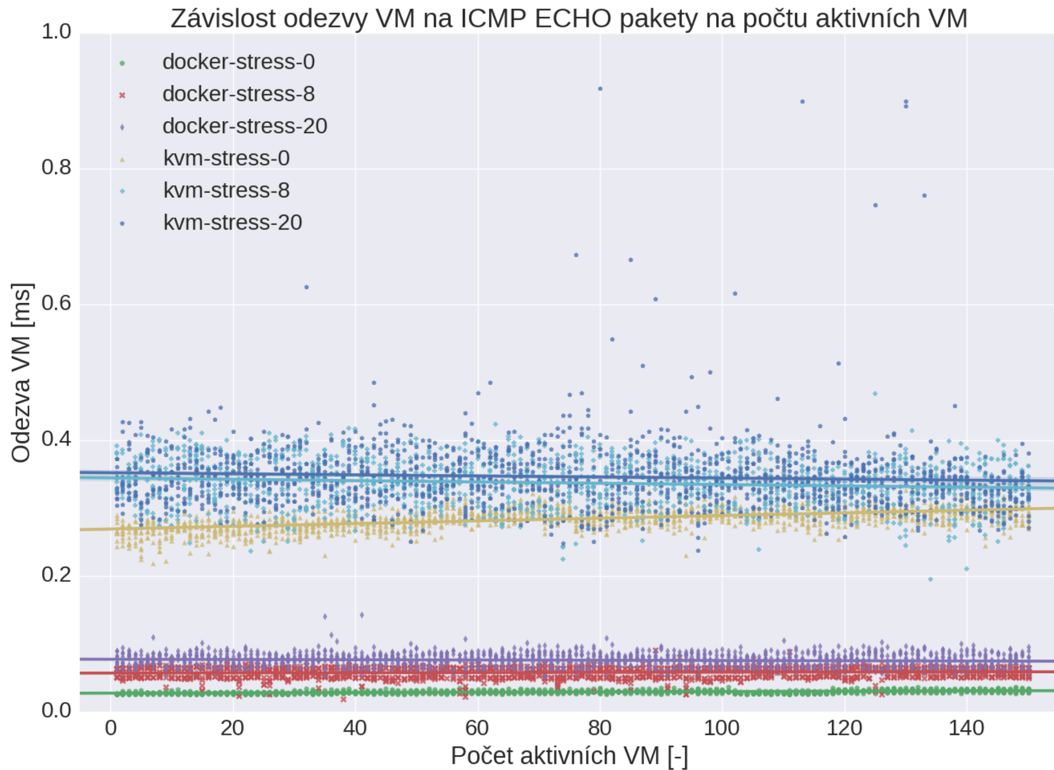
Obr. 3.2: Graf zátěže systému v intervalu 15 min při jednotlivých měřeních



Obr. 3.3: Graf využití operační paměti při jednotlivých měřeních

počtu využití paměti cca 9%, u KVM je při stejném počtu aktivních VM využito téměř 80% operační paměti. Tento markantní rozdíl je způsoben rozdílným přístupem obou technologií – zatímco VM u technologie KVM jsou „plnohodnotným“ systémem, který obsahuje i vlastní jádro (kernel), které musí být zavedeno do virtuální paměti RAM a s hostitelským systémem nesdílí nic kromě virtuálního HW, v rámci Docker kontejnerů sdílí spuštěné procesy jádro s hostitelem, v rámci kterého jsou izolovány od procesů hostitele i procesů v ostatních kontejnerech.

Virtuální stanice na KVM mají v tomto případě přiděleno 512 MB RAM, takže při maximálním počtu 150 aktivních VM by mělo být využito 75 GB operační paměti, přičemž hostitelský systém má pouze 48 GB paměti RAM. V tomto případě se uplatňuje technika zvaná `memory overcommit` (volně přeloženo jako nadměrné přidělení paměti), kdy se počítá s faktem, že ne všechny VM využívají neustále celou přidělenou operační paměť. Při použití této techniky je nutná opatrnost, protože v případě, kdy začne více VM ve stejný okamžik využívat více paměti, může dojít k vyčerpání paměti a zpomalení nebo kolapsu systému.



Obr. 3.4: Síťová odezva VM

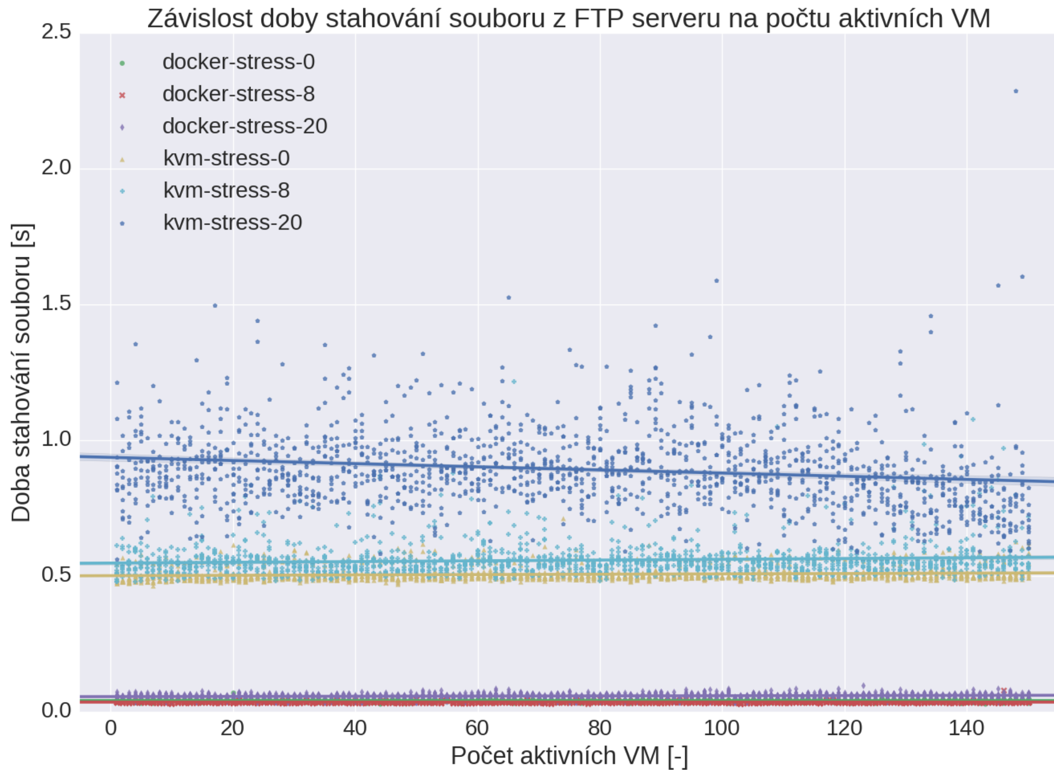
3.1.2 Ping

Na obrázku 3.4 je zobrazena závislost dalšího z měřených parametrů, a to síťové odezvy (latence) VM na počtu aktivních VM. Naměřené hodnoty se pohybují pod hodnotou 0,5ms, což je vzhledem k faktu, že měření probíhá na stejném serveru, a jednotlivé pakety jsou tedy přenášeny pouze v rámci virtuálních rozhraní, odpovídající hodnota. U KVM dosahují hodnoty latence přibližně 3-násobku hodnot naměřených u Docker kontejnerů, pravděpodobně z toho důvodu, že jsou pakety přenášeny a zpracovány ještě jedním virtuálním kernelem. Při vyšších úrovních zatížení serveru hodnoty latence mírně stoupají zejména díky většímu využití CPU, které potom pomaleji obsluhuje hardwarová přerušení (IRQ).

3.1.3 FTP

Dalším ze sledovaných parametrů byla doba stažení a nahrání souboru na vzdálený FTP server, spuštěný ve VM. V obou případech se jednalo o soubor o velikosti 10 MB. Na obrázku 3.5 si můžeme prohlédnout naměřené závislosti doby, kterou trvá stažení souboru, v závislosti na počtu aktivních VM a zatížení systému.

V případě virtualizace Docker vidíme, že naměřené hodnoty pro všechny úrovně zátěže a různé počty aktivních VM/kontejnerů jsou prakticky konstantní a pohybují



Obr. 3.5: Doba stahování souboru z FTP serveru pro jednotlivé virtualizace

se kolem 40 ms. V případě KVM jsou ale naměřené hodnoty bez zátěže vyšší téměř o 500 ms, a v případě, že je systém vytížen na cca 80%, se čas stahování souboru blíží i k jedné vteřině.

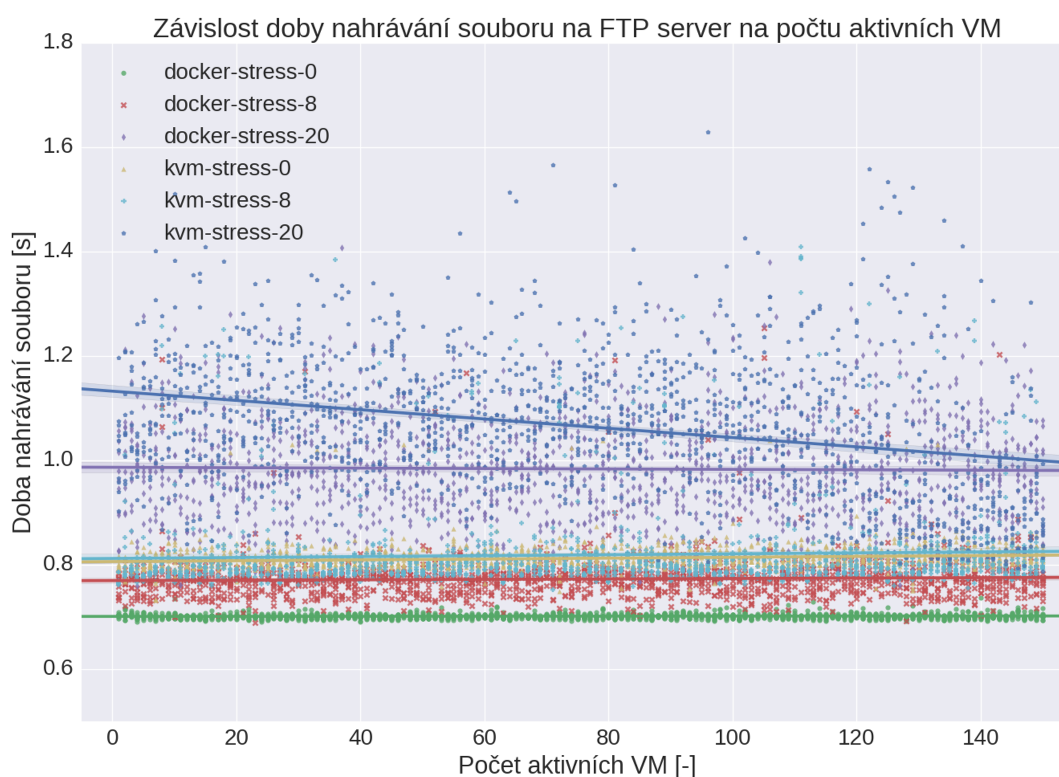
Na obrázku 3.6 je graf zobrazující závislost doby nahrávání souborů na FTP server. U virtualizace Docker se na systému bez zátěže pohybují kolem 700 ms, hodnoty na zatíženém systému poté kolem 1200ms, přičemž jsou tyto hodnoty srovnatelné s virtualizací KVM pro jednotlivé dvojice měření při stejné zátěži. Analýzou síťového provozu na virtuálních síťových rozhraních během měření byl proveden rozbor spojení, jehož výsledky jsou uvedeny v tabulce 3.2.

Čas, který trvají jednotlivé akce uvedené v tabulce 3.2, byl vypočítán následovně:

1. TCP handshake – rozdíl časových značek prvního TCP segmentu obsahujícího příznak SYN pro sestavení spojení směrem na FTP server, a TCP segmentu obsahujícího příznak ACK, který potvrzuje navázání spojení.
2. FTP – přihlášení – rozdíl časových značek FTP datagramu obsahujícího návratový kód FTP serveru „220: Service ready for new user“ a zasláním FTP příkazu PASS ze strany klienta.
3. DNS – reverzní záznam – protože použitý server vsFTPd během přihlášení klienta hledá reverzní DNS záznam pro jeho IP adresu, je v tabulce uvedena i tato akce. Jedná se o rozdíl časových značek datagramu obsahujícího první

Tab. 3.2: Analýza měření protokolu FTP

virtualizace - zátěž	**KVM - 0**	**Docker - 0**		**Docker - 20**
Upload / Download	**Down**		**Up**	
akce	**čas [ms]**			
TCP handshake	1	1	1	1
FTP - přihlášení	26	1	1	1
DNS - reverz. zázn.	1	25	25	30
FTP - příprava	50	5	646 + 2	1118 + 1
FTP - přenos	504	8	16	56
FTP - ukončení	1	1	20	72
Celkem	**573**	**41**	**711**	**1279**
Měření	**591**	**42**	**718**	**1293**



Obr. 3.6: Doba nahrání souboru na FTP server

DNS dotaz typu PTR, a obdržením finální DNS odpovědi.

4. FTP – příprava – Tento interval vyjadřuje čas, který uplynul od zaslání FTP kódu „230 Login successful“ klientovi a mezi posledním zachyceným paketem před zahájením přenosu souboru. V rámci tohoto intervalu byly ve většině měření provedeny následující akce:
   - vyžádání pasivního režimu klientem pomocí FTP příkazu PASV,
   - potvrzení přepnutí serverem odpovědí „227 Entering Passive Mode“,

včetně zaslání portu, na kterém bude server naslouchat příchozímu datovému spojení,
  - přepnutí do textového režimu pomocí příkazu TYPE A,
  - potvrzení přepnutí do textového režimu odpovědí „200 Switching to ASCII mode“,
  - získání seznamu souborů a jejich velikostí klientem pomocí příkazů NLST a SIZE,
  - přepnutí režimu spojení do binárního módu před přenosem binárního souboru (příkaz TYPE I, odpověď „200 Switching to binary mode“).
5. FTP – přenos – jedná se o rozdíl časové značky FTP datagramu obsahujícího příkaz RETR <název souboru> zaslaného serveru, a FTP datagramu obsahujícího odpověď serveru „260 Transfer Complete“.
6. FTP – ukončení – časový interval, který trvá ukončení spojení (FTP příkaz QUIT, odpověď 221 Goodbye, a ukončení TCP spojení).

V řádku „Celkem“ je potom uveden součet těchto intervalů získaných analýzou síťového provozu a v posledním řádku tabulky, „Měření“, je uvedena hodnota, která byla naměřena při stejném pokusu aplikací `vte`.

Při měření doby stahování souboru je u obou virtualizací nejmarkantnější rozdíl v intervalu, kde probíhá FTP přenos. Dodatečnou analýzou síťového provozu bylo zjištěno, že tento rozdíl vzniká díky velikosti datagramů obsahujících samotná FTP data. Zatímco datagramy zachycené při použití virtualizace KVM mají všechny shodnou velikost 1448 bajtů, což odpovídá standardní velikosti MTU[1], v případě Dockeru mají datagramy velikost až 65160 bajtů, což je maximální velikost přípustná IP protokolem. V jednom datagramu je tedy se stejnými režijními náklady přeneseno až 45krát více dat.

Dalším rozdílem je doba, kterou trvá vyhledání reverzního DNS záznamu. V tomto případě je rozdíl způsoben tím, že výchozí síťové rozhraní u virtualizace Docker poskytuje kontejnerům přístup do vnějších sítí pomocí překladu adres NAT, a dále kontejnerům v rámci konfigurace DHCP předávají DNS servery, které používá hostitelský systém. U KVM bylo jako DNS server použito virtuální rozhraní, takže zde byl DNS dotaz okamžitě vyřízen, zatímco v případě Dockeru DNS dotaz vyřizoval reálný DNS server. Další zpoždění u virtualizace Docker pak způsobily další DNS dotazy typů A a AAAA na záznam „gateway“.

V případě měření rychlosti nahrávání souboru na FTP server byly naměřené hodnoty u virtualizací KVM a Docker podobné, byl tedy analyzován pouze síťový provoz pro virtualizaci Docker při zatížení systému na 0% a 80%. Oproti stahování souboru činí rozdíl na systému bez zatížení přibližně 700 ms. Největší část tohoto

[1] Maximum Transmission Unit.

rozdílu způsobila prodleva na straně klienta mezi obdržením informace od serveru, že byl klient úspěšně přihlášen, a zasláním dalšího FTP příkazu, která je zachycena na obrázku 3.7

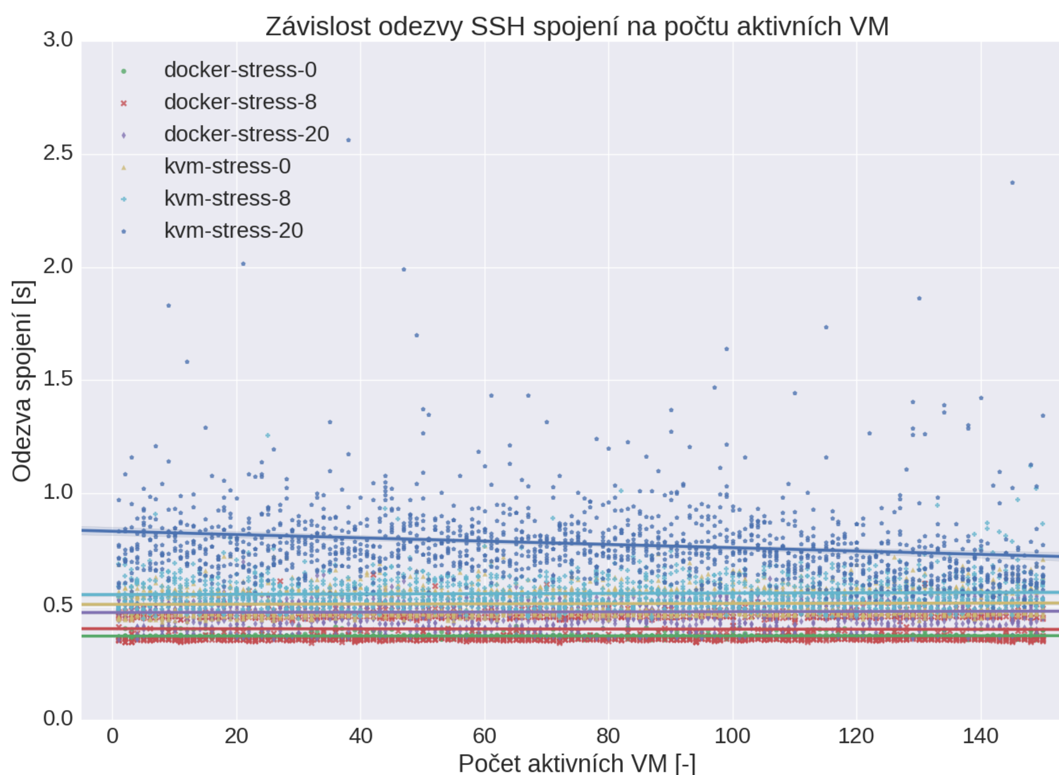
36	10.8224	172.17.0.2	172.17.0.1	FTP	Response: 230 Login successful.
37	10.8599	172.17.0.1	172.17.0.2	TCP	52177 > ftp [ACK] Seq=23 Ack=78 Win=29312
38	11.4680	172.17.0.1	172.17.0.2	FTP	Request: TYPE I
39	11.4682	172.17.0.2	172.17.0.1	FTP	Response: 200 Switching to Binary mode.

Obr. 3.7: Prodleva na straně klienta při nahrávání souboru na FTP server

Tato prodleva je v tabulce 3.2 zaznamenána v řádku „FTP – příprava“, kde se jedná o první hodnotu před symbolem „+“. Při zatížení systému pak tato prodleva vzroste na téměř dvojnásobnou hodnotu, což opět činí největší rozdíl v časových intervalech.

3.1.4 SSH

Další službou, jejíž charakteristiky byly měřeny, bylo vzdálené připojení pomocí služby SSH. Během měření byl vyhodnocován čas, který trvá sestavení spojení, provedení příkazu `uptime` na vzdálené stanici, a následně ukončení spojení. Výsledky tohoto měření jsou na obrázku 3.8.



Obr. 3.8: Graf naměřených hodnot pro službu SSH

Z hodnot na zmíněném obrázku 3.8 vidíme, že v případě použití virtualizace Docker byly při všech úrovních zátěže systému naměřeny i pro maximální počet VM hodnoty pod 500 ms. Pro virtualizace KVM vychází průměrná doba celého spojení na 500 ms bez zátěže, a přibližně na 800 ms se zátěží, přičemž některé naměřené hodnoty přesahují dobu 1 sekundy. Pro měření při zátěži systému 80% je také možné pozorovat, že naměřené hodnoty s rostoucím počtem aktivních VM mírně klesají, což je oproti předpokladu přesně opačné chování. Analýzou získaných dat se ale bohužel nepodařilo určit příčinu tohoto trendu.

Stejně jako u ostatních měření, i v tomto případě bylo provedeno několik izolovaných testů, kdy byl zároveň zachytáván síťový provoz pro další analýzu. Kvůli povaze protokolu SSH (data jsou přenášena šifrovaně) ale není možné analyzovat obsah jednotlivých datagramů, tudíž byla provedena pouze základní analýza. Následující rozbor se vztahuje k virtualizaci KVM, kde byl rozdíl mezi hodnotami naměřenými bez a se zátěží výraznější.

Při měření aplikací `vte` byla na nezatíženém systému doba trvání spojení určena na 606 ms. Z této doby se podařilo blíže identifikovat následují intervaly:

- 20 ms – prodleva po TCP handshake,
- 59 ms – doba trvání výměny klíčů a sestavení šifrovaného kanálu,
- 30 ms – prodleva po vyhledání reverzního DNS záznamu stanice.

Měření na systému vytíženém na 80% trvalo celkem 1,447 sekundy, z čehož se podařilo identifikovat následující intervaly:

- 80 ms – prodleva po TCP handshake,
- 72 ms – výměna klíčů a sestavení šifrovaného kanálu,
- 280 ms – prodleva před vyhledáním reverzního DNS záznamu,
- 320 ms – prodleva po vyhledání reverzního DNS záznamu.

3.1.5 Telnet

Poslední z testovaných služeb bylo vzdálené připojení pomocí služby Telnet, kde byl měřen čas, který trvá sestavení spojení, vykonání příkazu `uptime` na vzdálené stanici, který po zavolání vrací statistiky doby provozu systému a jeho zátěže, a následně ukončení spojení.

Na obrázku 3.9 jsou zobrazena naměřená a zaznamenaná data. Při použití Docker kontejnerů jsou naměřené hodnoty prakticky konstantní pro libovolný počet aktivních kontejnerů, a dosahují přibližně 150 milisekund při všech úrovních zatížení systému. U KVM je situace výrazně horší, při nulovém zatížení systému se průměrné hodnoty pohybují kolem hranice 350 ms, při 80% zatížení pak průměrně dvojnásobku (700 ms), při některých měřeních pak i přes 1 sekundu. Analýzou síťového provozu na virtuálních rozhraních při probíhajícím měření bylo zjištěno následující:

- **Docker – zatížení 0%**.

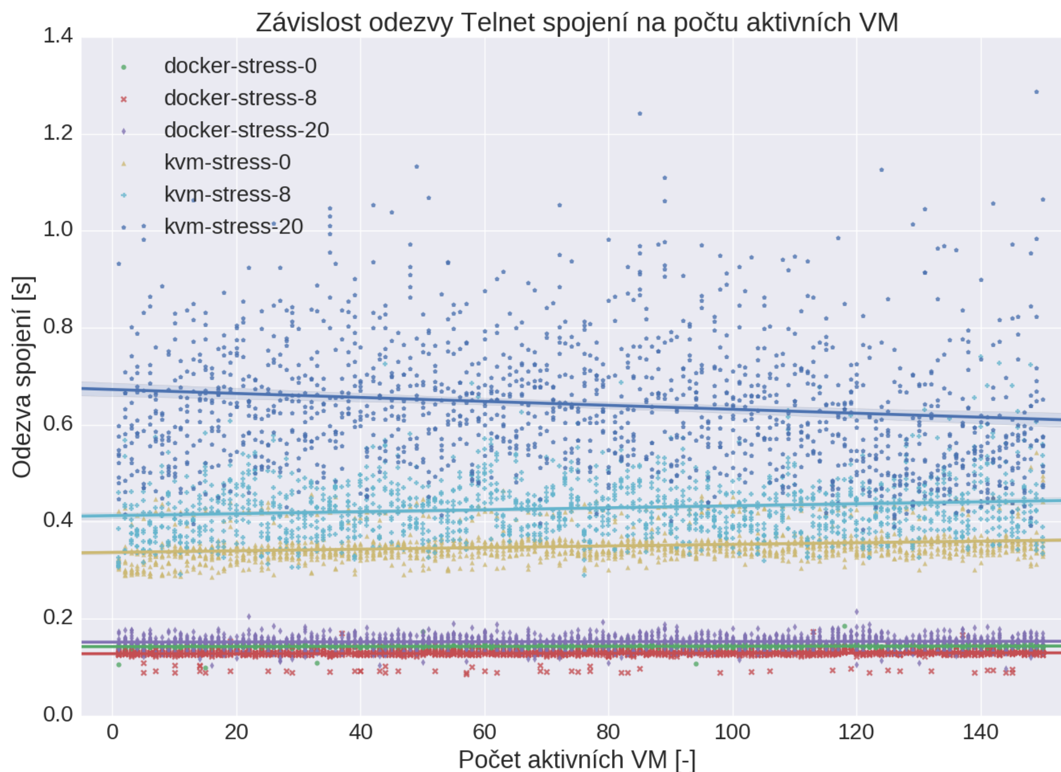
  Při provedení jednoho izolovaného měření se spuštěným zachytáváním síťového provozu aplikace `vte` naměřila hodnotu 215 ms. Následnou analýzou provozu bylo zjištěno, že samotné sestavení spojení a přihlášení zabralo z tohoto času přibližně 2 ms, provedení příkazu `uptime` na vzdálené stanici 11 ms. 13 ms potom zabralo vyhledávání reverzního DNS záznamu pro adresu virtuálního rozhraní a několik DNS dotazů na záznam „gateway". Zbytek doby byla vzdálená stanice nečinná, tzn. mezi jednotlivými zachycenými pakety docházelo k významných prodlevám (mezi největší patří např. prodleva 49 ms po sestavení Telnet spojení a zasláním výzvy pro přihlášení, nebo prodleva 45 ms mezi zasláním informace o době posledního přihlášení a zobrazení příkazového řádku).
- **KVM – zatížení 0%**.

  U měření na virtualizaci KVM připadá, stejně jako v předchozím případě, nejvíce z naměřené doby na prodlevy mezi jednotlivými akcemi, které jsou u této virtualizace ještě markantnější, než v případě virtualizace Docker. Celková naměřená doba činí 484 ms, nově např. došlo k prodlevě v trvání 40 ms mezi TCP handshake a začátkem sestavování Telnet spojení, další prodlevy se pak vyskytly mezi zasláním informace o posledním přihlášení a zobrazení příkazového řádku (81 ms), resp. mezi posledním TCP potvrzením ACK a zobrazením informace o přihlášení (180 ms). Samotné vykonání příkazu `uptime` poté trvalo 16 ms, a vyhledání reverzního DNS záznamu pouze 1ms.
- **KVM – zatížení 80%**.

  V tomto případě trvalo izolované měření celkem 1,276 s, tedy zhruba třikrát déle, než na systému bez zátěže. Největší část zabraly následující položky:
  - 530 ms – prodleva mezi zprávou TCP ACK zaslanou z virt. rozhraní a zobrazením zprávy o čase posledního přihlášení,
  - 410 ms – prodleva mezi předchozí zprávou a zobrazením příkazové řádky,
  - 91 ms – prodleva mezi TCP handshake a začátkem Telnet spojení,
  - 60 ms – prodleva mezi ověřením uživatelského jména a zasláním výzvy pro zadání hesla,
  - 40 ms – součet prodlev během sestavování Telnet spojení,
  - 39 ms – vykonání příkazu `uptime`.

3.1.6 Zhodnocení

Při porovnání obou popisovaných nástrojů, resp. odlišných principů, na kterých Docker a KVM pracují, vychází na základě naměřených hodnot a analýz popsaných v předchozích podkapitolách virtualizace aplikací pomocí kontejnerů z výkonnost-



Obr. 3.9: Naměřené hodnoty pro službu Telnet

ního hlediska podstatně lépe, než tradiční softwarová virtualizace. Největší výhodou virtualizovaných kontejnerů je fakt, že sdílí kernel s hostitelem, čímž jsou jednak eliminovány režijní náklady na provoz dalších virtuálních kernelů (zejména paměti RAM), a dále má tento fakt pozitivní vliv na propustnost sítě (pakety není třeba zpracovávat v rámci virtuálních síťových zařízení a dále je předávat ke zpracování virtuálnímu kernelu), což se pozitivně odráží i na výkonosti síťových služeb. Zejména zmíněná úspora operační paměti RAM je při větším počtu aktivních VM velmi znatelná, jak bylo popsáno v kapitole 3.1.1.

Další z výhod kontejnerové virtualizace, resp. konkrétně její implementace v rámci nástroje Docker, je úspora místa na disku z důvodu použití vrstvového souborového systému, kdy je možné sdílet stejný základ více kontejnery a následně nad tímto základem stavět další vrstvy. Zatímco obrazy virtuálních disků použité při tomto měření pro jednotlivé VM na KVM zabírají celkem cca 52 GB diskového prostoru, celkový objem místa zabraného virtuálním souborovým systémem **všech** Docker kontejnerů je celkem 441 MB.

Závěrem je ovšem nutné podotknout, že virtualizace aplikací pomocí kontejnerů neposkytuje díky sdílenému kernelu tak silnou izolaci od systému hostitele, jako tradiční VM, a není také zcela vhodná pro všechna použití (typicky např. pro monolitické aplikace vyžadující velké množství zdrojů, které se nedají škálovat do

šířky).

3.2 Výkon nástrojů Docker a KVM/QEMU při zatížení systému

Kromě vzájemného porovnání virtualizačních nástrojů Docker a KVM při různých úrovních zatížení všech částí systému bylo také provedeno několik měření, v rámci kterých byl analyzován vliv izolovaného zatížení jedné z částí systému (CPU, paměť, disky) na provoz virtuálních stanic u daného nástroje. Kvůli velkému počtu provedených měření byly do práce zařazeny pouze vybrané grafy (zejména pro služby, kde se při zatížení dané části systému projevil velký výkonnostní rozdíl).

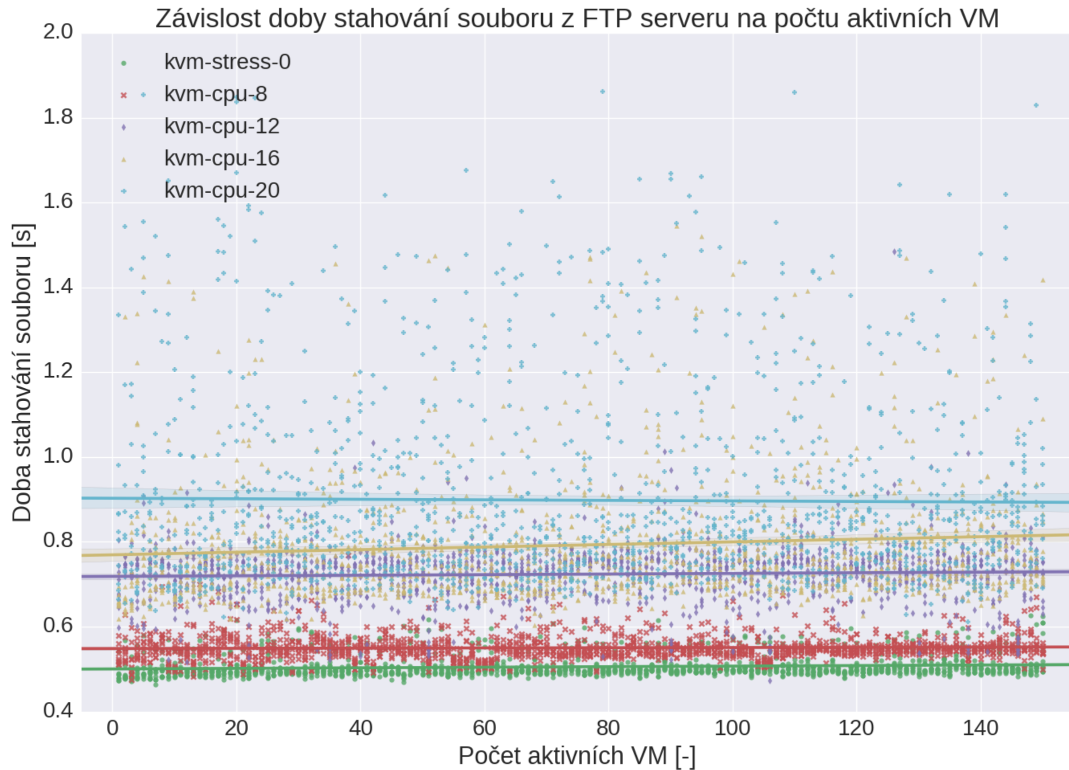
3.2.1 Výkon při zatížení CPU

Na obrázku 3.10 je zobrazena závislost doby stahování souboru z FTP serveru na zatížení procesoru pro nástroj KVM. Ze závislostí vidíme, že zatímco počet aktivních VM nemá na měřené hodnoty prakticky žádný vliv, rostoucí zatížení procesoru způsobuje nárůst doby stahování souboru poměrně významným způsobem. Dále je vidět, že při zatížení procesoru přes 80% dochází k velkému rozptylu měřených hodnot.

Na obrázku 3.11 je zobrazena závislost doby trvání SSH spojení na zatížení procesoru pro virtualizaci KVM, stejně, jako v předchozím případě. V grafu můžeme opět vidět podobný trend, kdy počet aktivních VM nemá přílišný vliv na měřené hodnoty (s výjimkou měření při zatížení procesoru cca na 50%, kde je vzrůstající trend vidět), ale samotné zatížení procesoru hodnoty ovlivňuje již významně. Důvodem nárůstu hodnot je zvýšení počtu prodlev a jejich trvání mezi jednotlivými akcemi (popsáno v kap. 3.1.1).

Zatížení CPU má vliv také na výkonnost síťových služeb při použití virtualizace Docker. Na obrázku 3.12 je zobrazena závislost doby nahrávání souboru na FTP server v závislosti na počtu spuštěných Docker kontejnerů a úrovni zatížení CPU pomocí nástroje `stress`. Zatímco vliv počtu kontejnerů na výkon služby je zanedbatelný, při maximálním zatížení procesoru je rozdíl naměřených hodnot téměř dvojnásobný.

Na dalším obrázku, 3.13, je potom zobrazena stejná závislost pro službu SSH. Stejně jako v předchozím případě je vidět nárůst doby spojení na přibližně 1,5-násobek hodnot bez zátěže.



Obr. 3.10: KVM – Vliv zatížení CPU na dobu stahování souboru z FTP serveru

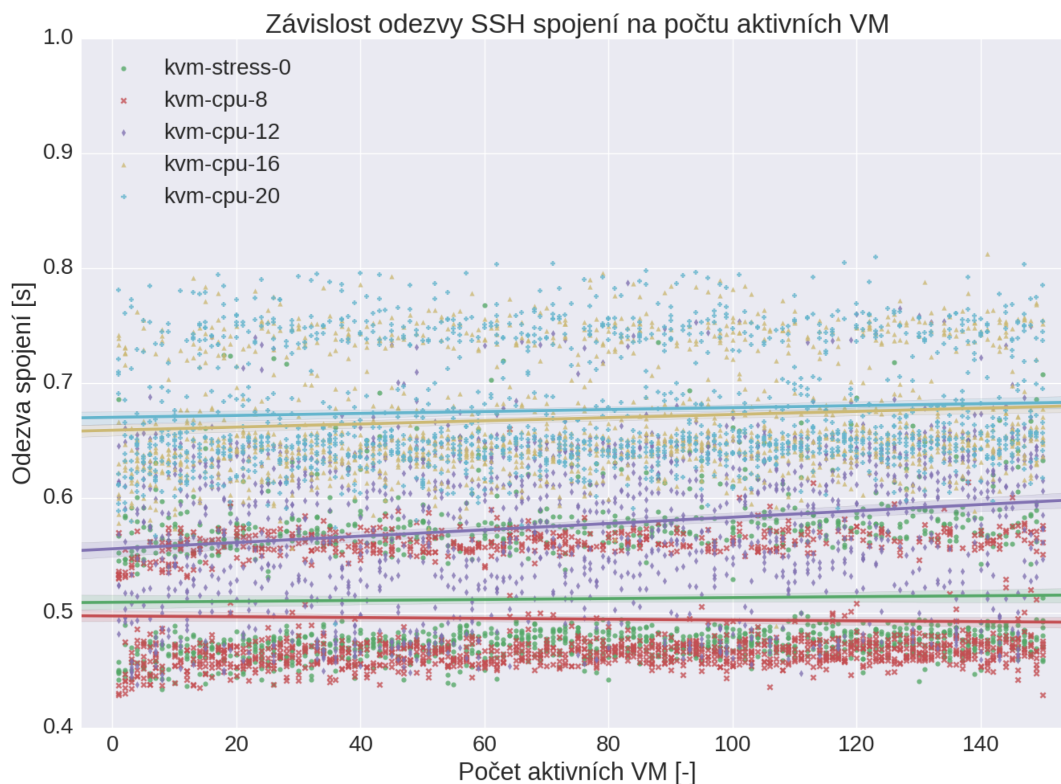
3.2.2 Výkon při zatížení operační paměti

Další měření byla provedená při postupném zvětšování využité paměti RAM. Při provádění tohoto měření se u virtualizace KVM podařilo úspěšně dokončit pouze testy při zatížení na 0% a cca 30% (tagy `kvm-stress-0` a `kvm-mem-8`). Ostatní měření postupně selhala, protože již nebylo možné alokovat další paměť RAM pro nově startované VM. Při měření s RAM vytíženou na 50% (tag `kvm-mem-12`) se podařilo naměřit hodnoty pro 131 aktivních VM, pro měření s vytížením cca 65% (tag `kvm-mem-16`) pro 102 VM, a pro měření se zatížením na cca 85% (tag `kvm-mem-20`) selhalo měření při pokusu o aktivování 89. VM. Ve všech případech skončilo měření kvůli chybě v knihovně libvirt oznamující, že se nezdařila alokace paměti:

```
libvirt: QEMU Driver error :  early end of file from monitor: possible problem:
Cannot set up guest memory 'pc.ram': Cannot allocate memory
```

Zátěž systému v té chvíli dosahovala ve všech případech hodnoty přes 100 a hostitelský server byl po přihlášení silně neresponzivní. Graf zátěže je zobrazen na obrázku 3.14, a využití paměti potom na obrázku 3.15.

Na obrázku 3.16 je poté zobrazen graf hodnot naměřených pro službu SSH, ze kterého je vidět, že při dosažení kritického bodu před selháním měření odezva spojení



Obr. 3.11: KVM – Vliv zatížení CPU na dobu trvání SSH spojení

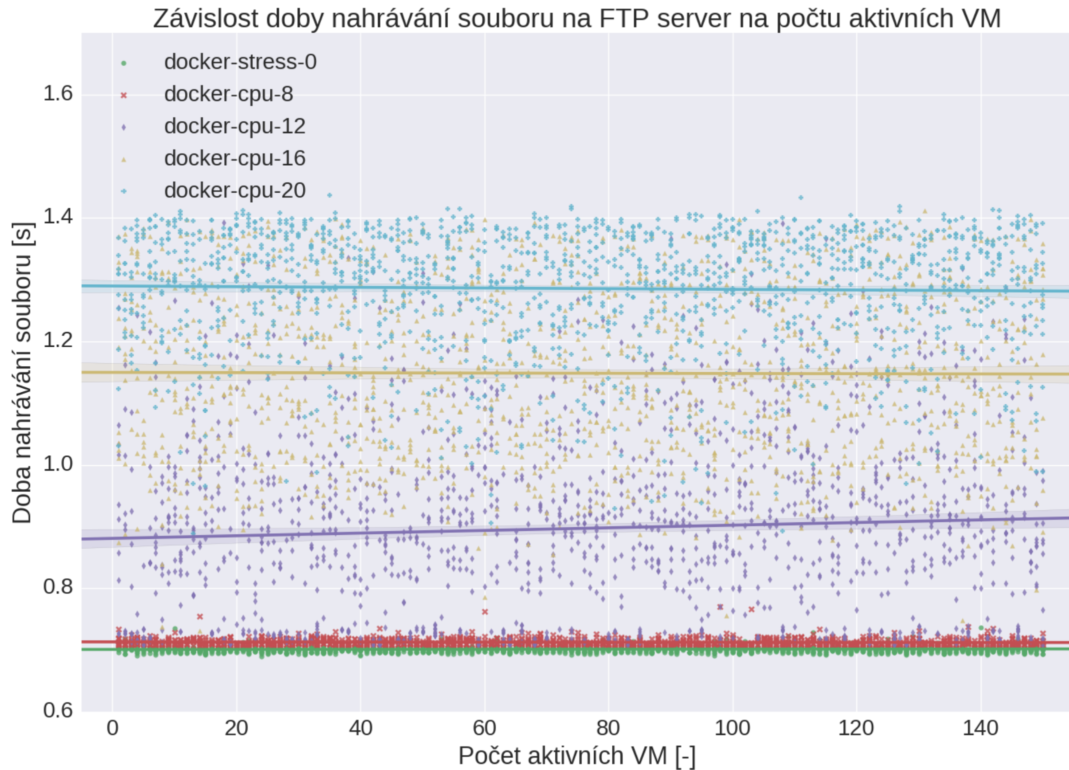
skokově vzrůstá až na cca 4-násobek doposud naměřených hodnot.

Naproti tomu proběhlo měření u virtualizace Docker bez problému hlavně díky faktu, že procesy spuštěné uvnitř kontejnerů jsou prakticky jen dalšími procesy v hostitelském systému, a není tedy třeba alokovat paměť pro virtuální jádro jako u virtualizace KVM. Na obrázku 3.17 je pro srovnání zobrazen graf hodnot naměřených pro službu SSH, kdy i hodnoty při maximálním zatížení systému dosahují přibližně polovičních hodnot oproti stejnému měření na KVM.

3.2.3 Výkon při zatížení IO

Poslední měření zkoumalo závislost výkonu síťových služeb při zatížení disků. Provedeno bylo pouze jedno měření se zátěží, protože nástroj `stress` vždy využil veškeré dostupné kapacity pro zápis i v případě, že byl spuštěn pouze v jednom vlákně. V případě, že bylo spuštěno více vláken zatěžujících HDD, byly kapacity rozděleny mezi tato vlákna, jak je možné vidět na ukázce 3.1, kde je zobrazen výstup z nástroje `iotop` během měření.

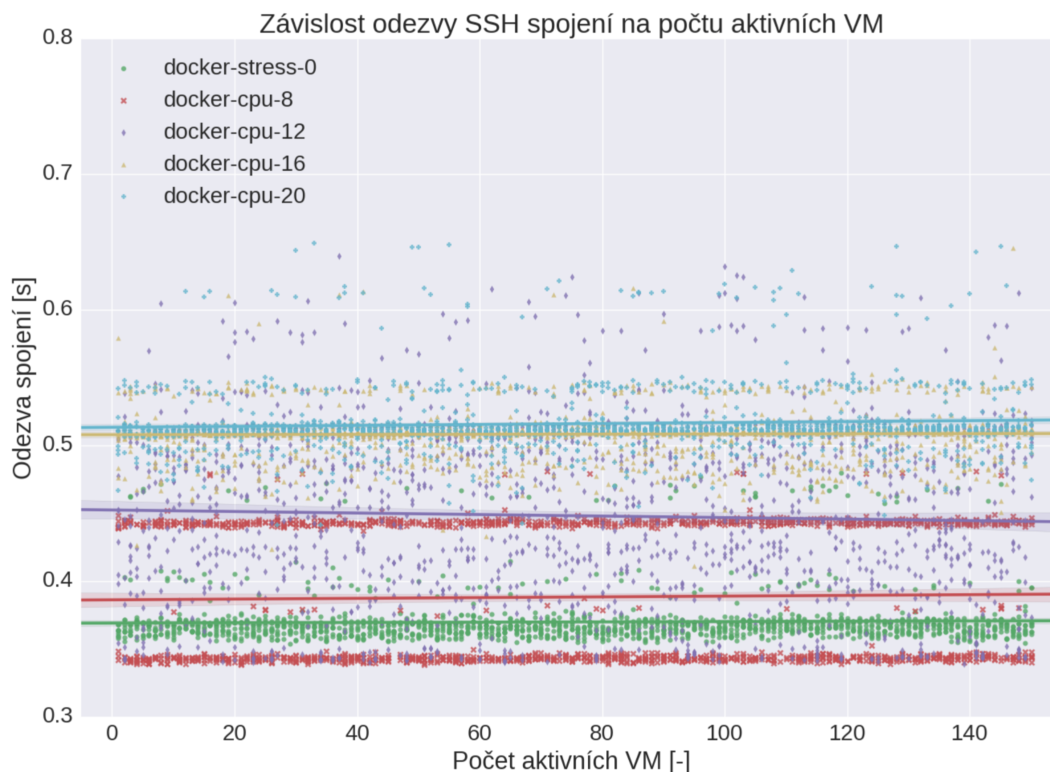
Podle zobrazeného výstupu bylo během měření neustále zapisováno na disky rychlostí kolem 700 MB/s, což se následně projevilo i v naměřených hodnotách. Největší vliv měla tato zátěž na dobu, kterou procesory strávily ve stavu `iowait`, tedy



Obr. 3.12: Docker – Doba nahrávání souboru na FTP server podle vytížení CPU

čekáním na dokončení vstupně-výstupních operací. Graf závislosti procesorového času na počtu aktivních VM je zobrazen na obrázku 3.18, ze kterého vidíme, že procesory celkem v tomto stavu strávily přibližně od 15% do 20% času v závislosti na počtu aktivních VM. Graf na obrázku byl naměřen pro virtualizaci KVM, nicméně jeho průběh je prakticky identický i pro nástroj Docker.

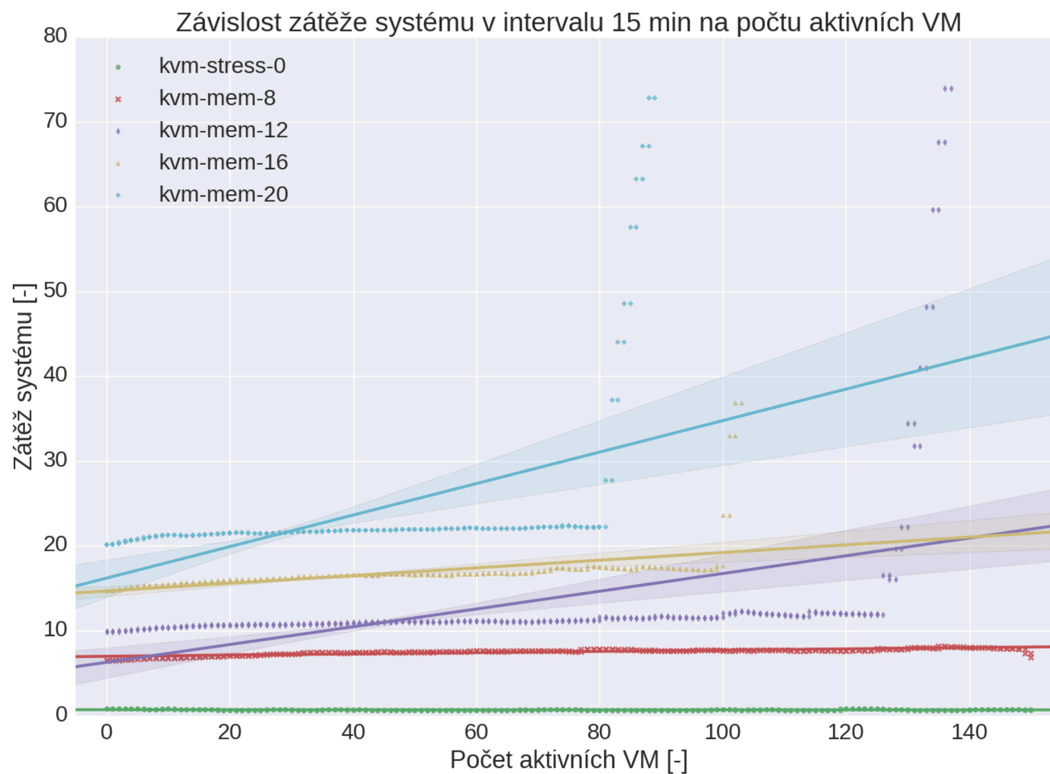
Na dalším obrázku 3.19 je potom zobrazena závislost doby stahování souboru z FTP serveru na počtu aktivních VM a zatížení disků pro virtualizaci KVM. Podle naměřených hodnot způsobilo zatížení disků při některých měřeních nárůst doby až na osminásobek oproti stavu bez zatížení. Hodnoty pro virtualizaci Docker jsou opět obdobné.



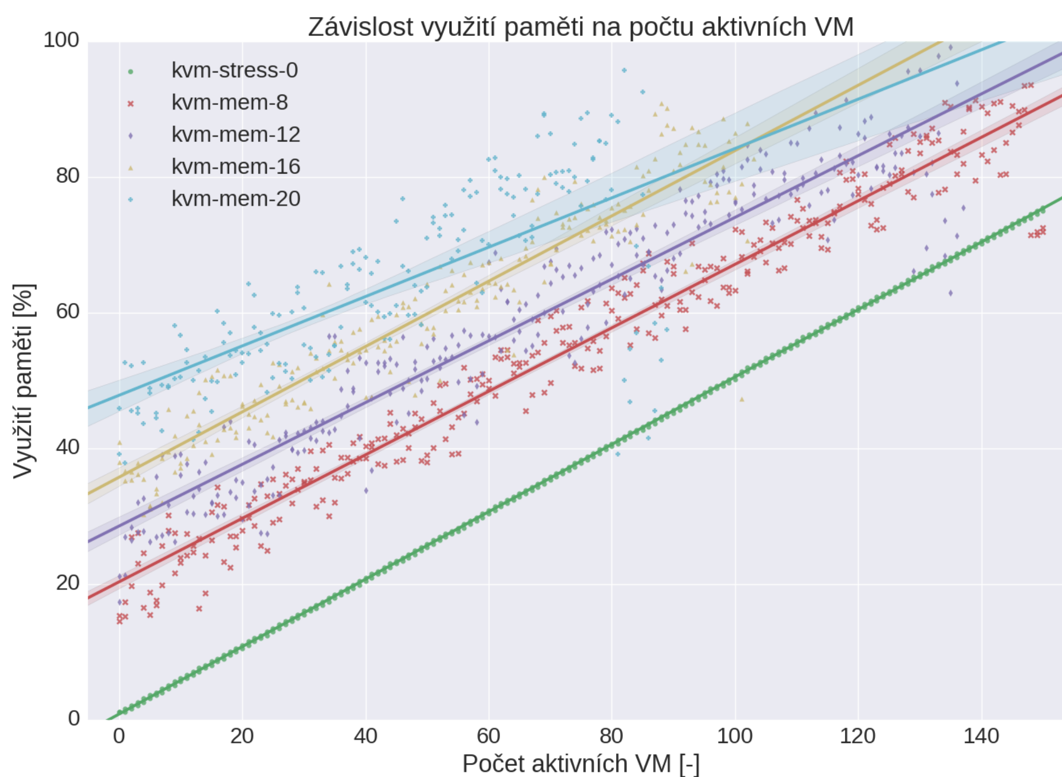
Obr. 3.13: Docker – Doba trvání SSH spojení podle vytížení CPU

```
Total DISK READ :      407.35 B/s | Total DISK WRITE :      697.75 M/s
Actual DISK READ:      407.35 B/s | Actual DISK WRITE:      678.82 M/s
  TID PRIO   DISK WRITE  SWAPIN PROC    COMMAND
23003 be/4   173.88 M/s  0.00 % 84.63 % stress -d 4 --hdd-bytes 10G -i 2
23001 be/4   174.52 M/s  0.00 % 84.57 % stress -d 4 --hdd-bytes 10G -i 2
23004 be/4   174.51 M/s  0.00 % 84.03 % stress -d 4 --hdd-bytes 10G -i 2
23005 be/4   174.82 M/s  0.00 % 81.87 % stress -d 4 --hdd-bytes 10G -i 2
```

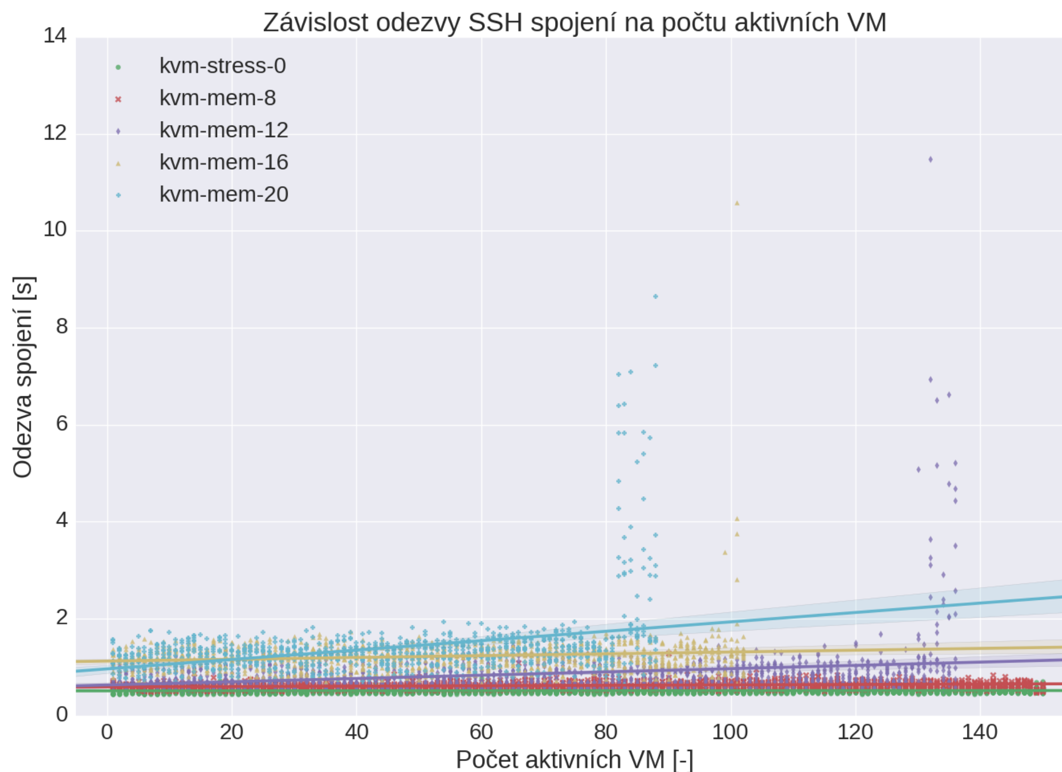
Ukázka 3.1: Část výstupu nástroje `iotop` během měření



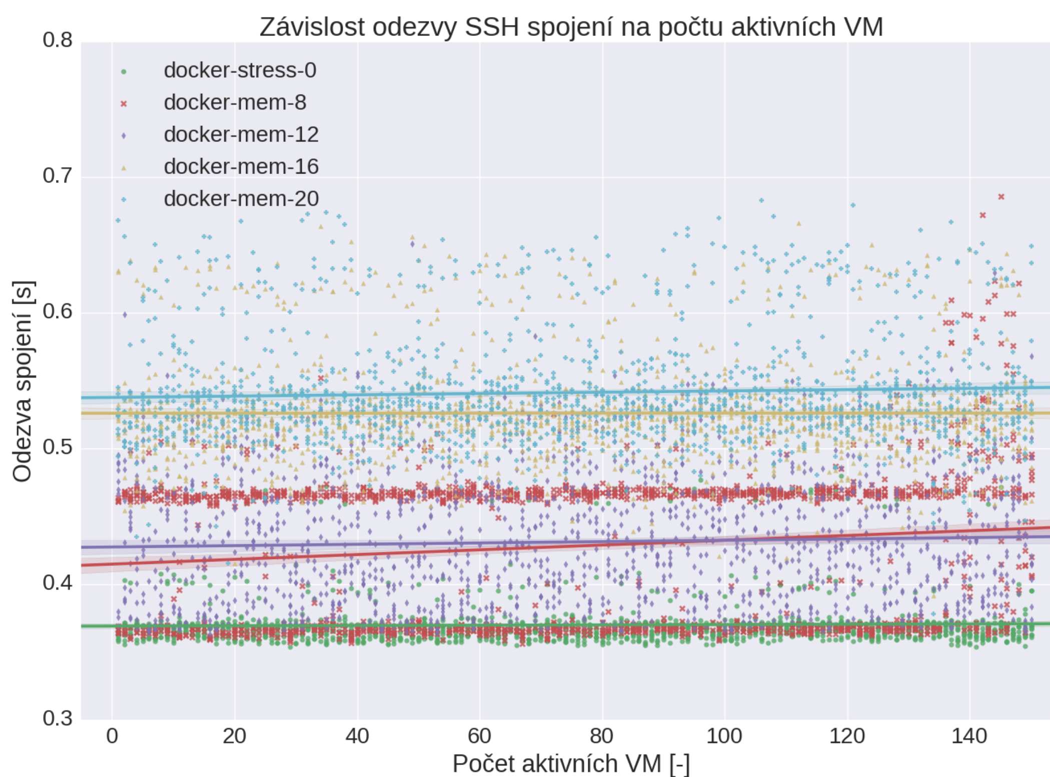
Obr. 3.14: KVM – Zátěž systému v závislosti na počtu aktivních VM a využité paměti RAM



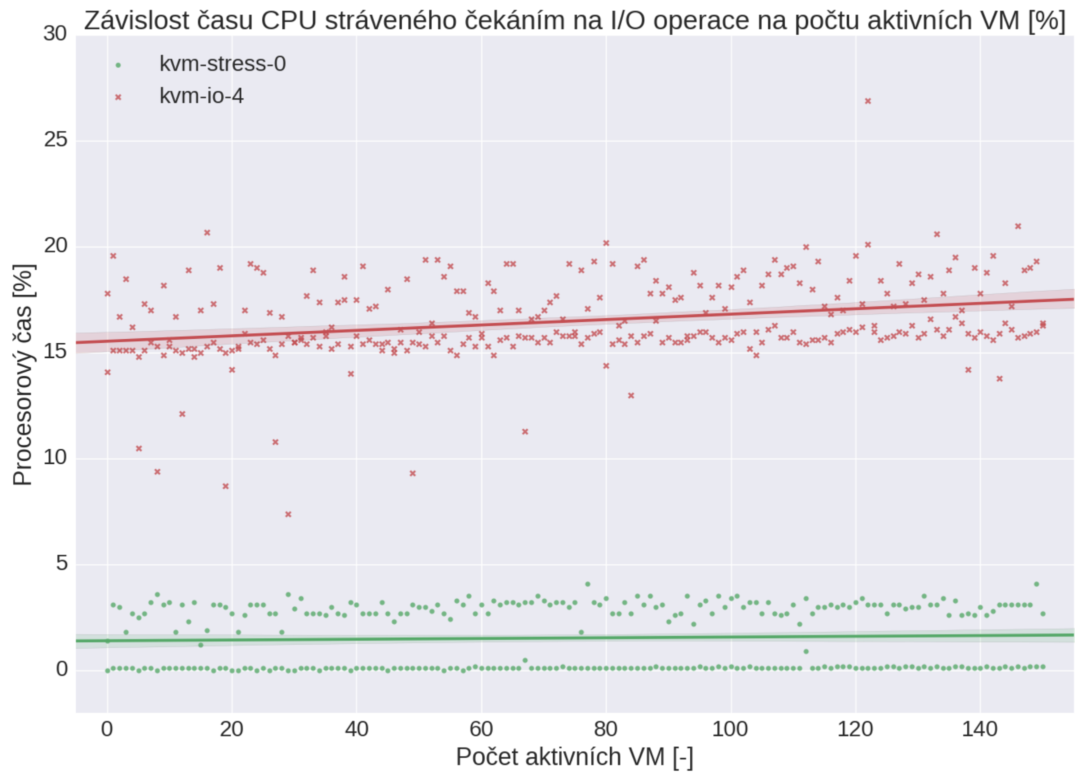
Obr. 3.15: KVM – Procento využité paměti RAM



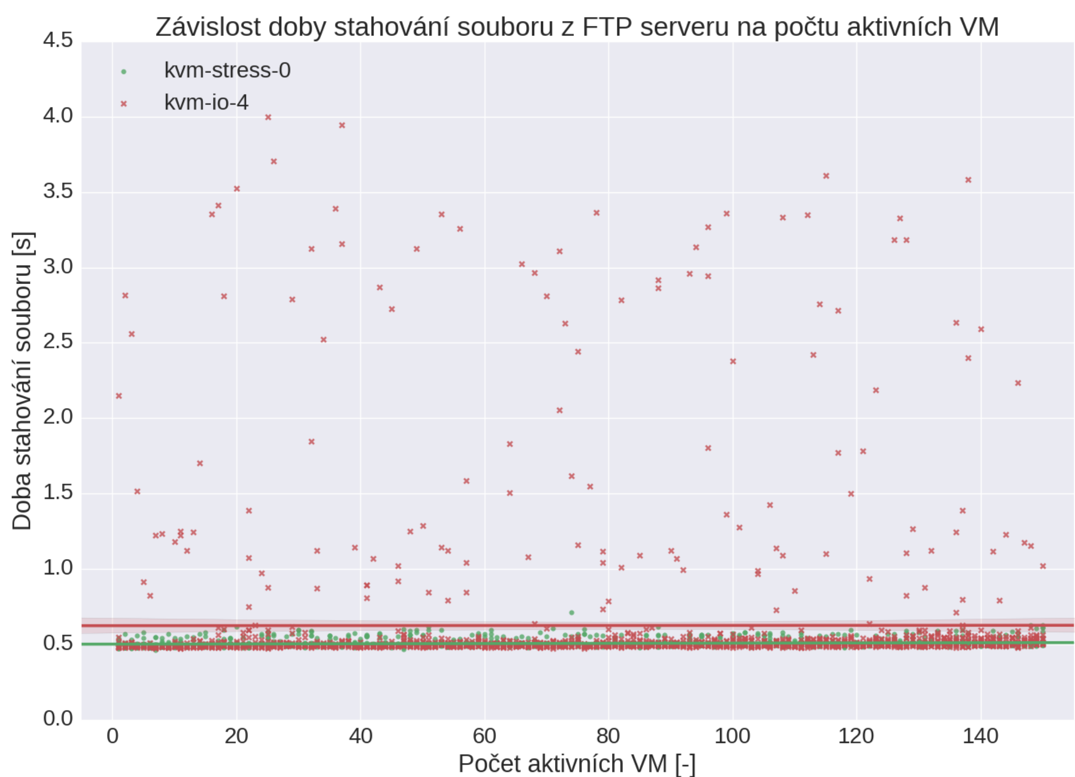
Obr. 3.16: KVM – Doba spojení SSH v závislosti na využité paměti a počtu aktivních VM



Obr. 3.17: Docker – doba trvání SSH spojení v závislosti na využití paměti a počtu aktivních VM



Obr. 3.18: KVM – Závislost procesorového času stráveného ve stavu `iowait` na počtu VM a zatížení disků



Obr. 3.19: KVM – Závislost doby stahování souboru z FTP serveru

4 ZÁVĚR

Tato práce se věnovala ověření parametrů několika vybraných síťových služeb (FTP, SSH, Telnet, Ping) provozovaných na virtualizovaných stanicích realizovaných pomocí virtualizačních nástrojů KVM a Docker. V rámci práce byly nejprve popsány jednotlivé typy virtualizace se zaměřením na KVM a Docker, a dále byly také rozebrány protokoly a služby, na jejichž parametry byla tato práce zaměřena. Součástí ověření bylo i měření a porovnání výkonu jednotlivých služeb při různém počtu aktivních VM a s různou úrovní zátěže uměle generované na serveru. Změřené závislosti parametrů zmíněných síťových služeb byly potom podrobně analyzovány.

V rámci realizace této práce byla vytvořena aplikace `vte`, která umožňuje měřit zmiňované parametry pro vybrané způsoby virtualizace automaticky, takže je možné měření provádět opakovaně s různými parametry pouze s minimálním úsilím. Vytvořená aplikace v současnosti obsahuje podporu pro měření parametrů na kontejnerové virtualizaci Docker a teoreticky na všech dalších virtualizačních nástrojích, pro které je implementována podpora v libvirt API. Aplikace je navíc modulární, takže je možné poměrně jednoduše rozšířit podporu o další typy virtualizace, či síťové služby.

Analýzou naměřených výsledků bylo zjištěno, že virtualizace má poměrně významný vliv na síťové služby. To je nejlépe vidět z rozdílů mezi hodnotami změřenými pro Docker kontejnery a tradiční VM na KVM, kdy u většiny naměřených závislostí dosahovaly virtuální stanice (resp. kontejnery) realizované pomocí nástroje Docker znatelně lepších hodnot, než srovnatelný počet virtuálních stanic při virtualizaci KVM. Tento rozdíl se pak ještě více prohloubil při extrémním zatížení serveru, na kterém bylo prováděno měření, kdy se nezdařilo část testů u virtualizace KVM dokončit kvůli nedostatku zdrojů v systému. Těmito měřeními bylo potvrzeno, že plnohodnotné virtuální stanice mají vyšší režijní náklady na provoz, než aplikační kontejnery, což se pak negativně projevuje i na výkonu provozovaných síťových služeb.

Závěrem je ale nutné dodat, že i když kontejnerová virtualizace dosahuje v tomto ohledu lepších výsledků, nemůže tradiční virtuální stanice zcela nahradit, např. typicky pro úlohy, kde je vyžadována striktní izolace hostitele a hostovaného systému, nebo v případě, kdy je na virtuálním systému nutné provozovat větší množství monolitických aplikací náročných na zdroje, které nelze zcela oddělit do jednotlivých kontejnerů a jednoduše škálovat do šířky.

LITERATURA

[1] Nanda, S.; Chiueh, T. *A Survey on Virtualization Technologies* [online]. Dept. of Computer Science, SUNY at Stony Brook, NY. [cit. 10. 12. 2015] Dostupné z URL: <http://www.ecsl.cs.sunysb.edu/tr/TR179.pdf>

[2] Popek, G. J.; Goldberg, R. P. *Formal requirements for virtualizable third generation architectures.* Communications of the ACM. Volume 17 Issue 7, July 1974. ISSN 0001-0782

[3] Anonym. *KVM - A small look inside* [online]. [cit. 10. 12. 2015]. Dostupné z URL: <http://www.linux-kvm.org/page/Small_look_inside>.

[4] Network World staff. *Desktop virtualization cheat sheet* [online]. [cit. 10. 12. 2015] Dostupné z URL: <http://www.networkworld.com/article/2237861/virtualization/desktop-virtualization-cheat-sheet.html>

[5] Matthias, K.; Kane, S. P. *Docker: Up and Running.* O'Reilly Media, Sebastopol, United States of America, 2015 ISBN 978-1-491-91757-2

[6] Anonym *Docker: What is Docker?* [online]. [cit. 10. 5. 2016] Dostupné z URL: <https://www.docker.com/what-docker>

[7] Anonym. *VMware® NSX for vSphere (NSX-V) Network Virtualization Design Guide* [online]. [cit. 10. 12. 2015] Dostupné z URL: <http://www.vmware.com/files/pdf/products/nsx/vmw-nsx-network-virtualization-design-guide.pdf>

[8] Rathod, H.; Townsend, J. *Virtualization 2.0 for Dummies.* John Wiley & Sons Ltd, Chichester, West Sussex, England, 2014 ISBN 978-1-119-02432-3

[9] Anonym. *libvirt Virtualization API - Terminology and Goals* [online]. [cit. 10. 12. 2015]. Dostupné z URL: <http://libvirt.org/goals.html>.

[10] Anonym. *libvirt Virtualization API - The virtualization API* [online]. [cit. 10. 12. 2015]. Dostupné z URL: <http://libvirt.org>.

[11] Anonym. *libvirt Virtualization API - Applications using libvirt* [online]. [cit. 10. 12. 2015]. Dostupné z URL: <http://libvirt.org/apps.html>.

[12] Huynh, K.; Hajnoczi, S. *KVM / QEMU Storage Stack Performance Discussion* [online]. [cit. 10. 5. 2016] Dostupné z URL: <http://www.ibm.com/support/knowledgecenter/api/content/nl/en-us/linuxonibm/liaav/LPCKVMSSPV2.1.pdf>

[13] Anonym *Docker Docs: Understand the architecture* [online]. [cit. 10. 5. 2016] Dostupné z URL: <https://docs.docker.com/engine/understanding-docker/>

[14] Anonym *ping(8) - Linux man page* [cit. 3. 5. 2016] Dostupné z URL: <http://linux.die.net/man/8/ping>

[15] Postel, J.; *RFC 792 - Internet Control Message Protocol*. Technická zpráva, Internet Engineering Task Force - Network Working Group, 1981

[16] Postel, J.; Reynolds, J. *RFC 959 - File transfer protocol (FTP)*. Technická zpráva, Internet Engineering Task Force - Network Working Group, 1985

[17] Klensin, J.; Hoenes, A. *RFC 5797 - FTP Command and Extension Registr.* Technická zpráva, Internet Engineering Task Force - Network Working Group, 2010

[18] Ylonen, T.; Lonvick, C; *RFC 4251 - The Secure Shell (SSH) Protocol Architecture*. Technická zpráva, Internet Engineering Task Force - Network Working Group, 2006

[19] Ylonen, T.; Lonvick, C; *RFC 4253- The Secure Shell (SSH) Transport Layer Protoco*. Technická zpráva, Internet Engineering Task Force - Network Working Group, 2006

[20] Ylonen, T.; Lonvick, C; *RFC 4252 - The Secure Shell (SSH) Authentication Protocol*. Technická zpráva, Internet Engineering Task Force - Network Working Group, 2006

[21] Ylonen, T.; Lonvick, C; *RFC 4254 - The Secure Shell (SSH) Connection Protocol*. Technická zpráva, Internet Engineering Task Force - Network Working Group, 2006

[22] Postel, J.; Reynolds, J. *RFC 854 - Telnet Protocol Specification*. Technická zpráva, Internet Engineering Task Force - Network Working Group, 1983

[23] Godard, S. *mpstat(1) - Linux man page*. [online]. [cit. 10. 5. 2016]. Dostupné z URL: <http://linux.die.net/man/1/mpstat>.

[24] Rodola, G. *psutil documentation*. [online]. [cit. 10. 5. 2016]. Dostupné z URL: <https://pythonhosted.org/psutil/#memory>.

[25] Greenfield, L.; Johnson, M. K. *uptime(1) - Linux man page*. [online]. [cit. 10. 5. 2016]. Dostupné z URL: <http://http://man7.org/linux/man-pages/man1/uptime.1.html>.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

ACK	Acknowledge
ACPI	Advanced Configuration and Power Interface
AES	Advanced Encryption Standard
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BSD	Berkeley Software Distribution
CLI	Command Line Interface
CPU	Central Processing Unit
CSV	Comma Separated Values
DB	Database
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
FTP	File Transfer Protocol
FTPS	File Transfer Protocol – Secured
GB	Gigabyte
GPG	GNU Privacy Guard
GSSAPI	Generic Security Services Application Program Interface
HDD	Hard Disk
HTML	HyperText Markup Language
HTTPS	Secure HyperText Markup Language
HW	Hardware
ICMP	Internet Control Message Protocol
IO	Input / Output
IP	Internet Protocol

IPv4 Internet Protocol version 4

ISO/OSI International Organization for Standardization / Open System Interconnection

KVM Kernel-based Virtual Machine

KVM/QEMU Kernel-based Virtual Machine / Quick Emulator

LTS Long Term Support

LXC Linux Containers

MAC Media Access Control

MB Megabyte

MS Microsoft

MTU Maximum Transmission Unit

NAT Network Address Translation

NVT Network Virtual Terminal

OS Operating System

QEMU Quick Emulator

RAID Redundant Array of Independent Disks

RAM Random Access Memory

RDS Remote Desktop Services

REST Representational State Transfer

RPM Revolutions per Minute

RSA Rivest, Shamir, Adleman

RTT Round Trip Time

SAS Serial Attached Storage

SMTP Simple Mail Transfer Protocol

SQL Structured Query Language

SSD	Solid State Disk
SSH	Secure Shell
SSL	Secure Sockets Layer
SYN	Synchronize
TCP	Transfer Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VLAN	Virtual Local Area Network
VM	Virtual Machine
VNC	Virtual Network Computing
WSGI	Web Services Gateway Interface
XML	Extended Markup Language

SEZNAM PŘÍLOH

A Obsah přiloženého DVD **93**

B Aplikace vte **94**

B.1 Instalace aplikace vte 94

B.2 Externí závislosti aplikace vte 95

B.2.1 Redis 95

B.2.2 Celery 96

B.3 Seznam hodnot výchozí konfigurace aplikace `vte` 96

A OBSAH PŘILOŽENÉHO DVD

Přiložený DVD nosič obsahuje následující soubory:

1. `centos-all.tar.gz` – Archiv obsahuje soubory potřebné k vytvoření výchozí Docker image popsané v kapitole 2.2.3.
2. `virtual-test-environment.tar.gz` – Archiv obsahuje soubory aplikace `vte`, jejíž instalace je popsána v příloze B.1.
3. `scripts.tar.gz` – Archiv obsahuje následující pomocné skripty:
   (a) `vm_install.py` – Skript pro vygenerování většího počtu QEMU VM (kap. 2.1.6).
   (b) `generate_network_config.py` – Skript pro vygenerování síťové konfigurace pro QEMU VM (kap. 2.1.2).
4. `data_csv.tar.gz` – Archiv obsahuje veškerá naměřená data převedená do záznamů CSV.
5. `centos-00.qcow2` – Výchozí obraz disku pro QEMU VM.

B APLIKACE VTE

V rámci této přílohy je popsána instalace aplikace `vte` ze souborů na přiloženém DVD nosiči a její výchozí parametry.

Aplikace byla testována a úspěšně provozována na systému s OS Debian 8.2 (jádro `3.16.7-ckt11-1+deb8u3 (2015-08-04) x86_64 GNU/Linux`), a verzi Python interpreteru `Python 3.4.2 (default, Oct 8 2014)`, mělo by jí ale být možné provozovat na libovolné verzi Pythonu od verze 3.3 a libovolném OS Linux za předpokladu, že jsou dostupné všechny potřebné systémové balíčky vypsané v kapitolách 2.1.1 a 2.2.1.

B.1 Instalace aplikace vte

Protože je aplikace `vte` napsaná v jazyce Python a je závislá na několika dalších balíčcích, je doporučeno ji nainstalovat do virtuálního přostředí (tzv. *virtualenv*), aby nedošlo ke kolizi se systémovými balíčky a Python knihovnami. Pro vytváření virtuálních přostředí je nutné mít v systému nainstalovaný balíček `python-virtualenv`.

Virtuální prostředí je poté možné vytvořit příkazem:

```
$ virtualenv -p python3.4 vte_venv
```

kde parametr `-p python3.4` určuje, která verze interpreteru má být ve virtuálním prostředí použita, a `vte_venv` je název cílové složky.

Virtuální prostředí se potom aktivuje příkazem:

```
$ source vte_venv/bin/activate
```

V případě, že je místo shellu `Bash` použit `Fish` nebo `csh`, je nutné použít odpovídající skript (`activate.fish` nebo `activate.csh`).

Po aktivaci virtuálního prostředí je možné aplikaci rozbalit a nainstalovat:

```
$ tar zxvf virtual-test-environment.tar.gz
$ cd virtual-test-environment
$ python setup.py install
```

Po instalaci by měla být aplikace dostupná z příkazové řádky (ale pouze s aktivovaným virtuálním prostředím) po zavolání příkazu `vte`.

Během instalace může dojít k následující chybě:

```
file build/libvirt_qemu.py (for module libvirt_qemu) not found
file build/libvirt_lxc.py (for module libvirt_lxc) not found
libvirt-override.c:25:27: fatal error: build/libvirt.h: No such file or directory
 #include "build/libvirt.h"
                           ^
compilation terminated.
error: command 'x86_64-linux-gnu-gcc' failed with exit status 1
```

V tom případě pravděpodobně není v systému nainstalován potřebný balíček `python3-libvirt`.

B.2 Externí závislosti aplikace vte

Aplikace `vte` je závislá na dvou externích aplikacích: Asynchronní fronta úloh `Celery` a úložiště hodnot `Redis`.

B.2.1 Redis

Aplikaci `Redis` lze stáhnout z oficiálních stránek, poslední verze dostupná během realizace této práce byla `http://download.redis.io/releases/redis-3.2.0.tar.gz`. [1] Po rozbalení archivu do cílové složky je nutné aplikaci zkompilovat pomocí příkazu `make`. Následně by ji mělo být možné spustit pomocí skriptu `./src/redis-server`, jak je zobrazeno na ukázce B.1

```
$ ./redis-server
3248:C 13 May 10:59:06.009 # Warning: no config file specified, using the default config.

    Redis 3.0.7 (00000000/0) 64 bit
    Running in standalone mode
    Port: 6379
    PID: 3248
    http://redis.io

3248:M 13 May 10:59:06.011 * DB loaded from disk: 0.000 seconds
3248:M 13 May 10:59:06.011 * The server is now ready to accept connections on port 6379
```

Ukázka B.1: Spuštění aplikace Redis

Aplikace Redis musí být spuštěná během celého měření, protože jsou do ní dočasně ukládána naměřená data před jejich finálním uložením do databáze.

[1] Dostupné k 10.5.2016

B.2.2 Celery

Dalším procesem, která musí být během meření spuštěn, je alespoň jeden `Celery worker` proces, který bude vykonávat úlohy generované aplikací `vte`. Aplikace `Celery` byla nainstalována jako závislost již během instalace aplikace `vte`, proto proces stačí jen spustit. Pro spuštění je nutné aktivovat virtuální prostředí s nainstalovanou aplikací `vte` kroky popsanými výše a proces spustit zadáním příkazu `celery -A vte.tasks worker`. Následně by se měl zobrazit podobný výpis jako na ukázce B.2.

```
(venv) $ celery -l info -c 1 -A vte.tasks worker

 -------------- celery@vte v3.1.20 (Cipater)
---- **** -----
--- * ***  * -- Linux-3.16.0-4-amd64-x86_64-with-debian-8.2
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> app:         app.tasks:0x7f57c31584a8
- ** ---------- .> transport:   redis://localhost:6379/0
- ** ---------- .> results:     redis://localhost/
- *** --- * --- .> concurrency: 1 (prefork)
-- ******* ----
--- ***** ----- [queues]
 -------------- .> celery       exchange=celery(direct) key=celery
[tasks]
  . vte.tasks.BaseTask
  . vte.tasks.FTPDownTask
  . vte.tasks.FTPUpTask
  . vte.tasks.PingTask
  . vte.tasks.SSHTask
  . vte.tasks.TelnetTask

[2016-05-13 11:08:43,971: INFO/MainProcess] Connected to redis://localhost:6379/0
[2016-05-13 11:08:43,980: INFO/MainProcess] mingle: searching for neighbors
[2016-05-13 11:08:44,997: INFO/MainProcess] mingle: all alone
[2016-05-13 11:08:45,008: WARNING/MainProcess] celery@vte ready.
```

Ukázka B.2: Spuštění procesu `celery worker`

B.3 Seznam hodnot výchozí konfigurace aplikace `vte`

Na ukázce B.3 je uveden seznam výchozích hodnot konfigurace aplikace `vte`. V případě, že je třeba některé z nastavení změnit, stačí v systému definovat proměnnou

prostředí s daným jménem a požadovanou hodnotou. Proměnné prostředí mají přednost před výchozími hodnotami definovanými v souboru `config.py`, který je popsán v kapitole 2.3.1.

```
TELNET_USERNAME = 'test'
TELNET_PASSWD = 'test'

FTP_USERNAME = 'test'
FTP_PASSWD = 'test'

SSH_USERNAME = 'test'
SSH_KEYFILE = '/home/kral/diplomka/docker/centos-all/id_rsa_test'

DOCKER_URL = 'http://127.0.0.1:2375'

LIBVIRT_URL = 'qemu:///system'
LIBVIRT_SUBNET = '10.255.255.0'
LIBVIRT_MAC_PREFIX = 'de:ad:be:ef:00:00'
LIBVIRT_LEASE_FILE = '/var/lib/libvirt/dnsmasq/net-kvm.leases'

LOGGING_LOGFILE = '/tmp/vte.log'

TASK_COUNTDOWN = 10
TASK_RETRY_COUNT = 5
```

Ukázka B.3: Výchozí konfigurace aplikace `vte`