VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZACE DOHLEDOVÉHO NÁSTROJE PRO SPRÁVU SÍTĚ VUT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. PETER DRIENKO
AUTHOR

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZACE DOHLEDOVÉHO NÁSTROJE PRO SPRÁVU SÍTĚ VUT

OPTIMIZING NETWORK TRACKING TOOL FOR MANAGEMENT OF BUT CAMPUS NETWORK

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. PETER DRIENKO
AUTHOR

VEDOUCÍ PRÁCE — Ing. TOMÁŠ PODERMAŃSKI
SUPERVISOR

BRNO 2014

Abstrakt

Tato diplomová práce se zabývá optimalizací monitorovacího nástroje Zabbix. V první části práce poskytuje přehled technik, kterými se monitorují počítačové sítě. Čtenáři objasňuje principy fungování nástroje Zabbix se zaměřením na monitorování velkého počtu zařízení. V druhé části práce jsou porovnány existující řešení optimalizace struktury databáze použitím rozdělování velkých, nadřazených tabulek na malé části. Dále je vybrané nejvhodnější řešení, které je upraveno pro potřeby VUT sítě. Díky této optimalizaci se snížilo zatížení procesoru Zabbix serveru a došlo k omezení růstu dat na pevném disku.

Abstract

This master's thesis focuses on optimizing the network tracking tool Zabbix. In the first part this thesis provides an overview of techniques used for monitoring computer networks. The thesis also describes the fundamentals of how Zabbix works, especially in large enviroments. In the second part of the thesis we compare existing optimizations of the database structure which divides large, parent tables into small partitions. The most suitable solution is chosen which is then modified for the BUT network needs. Using this optimization we achieved a decrease of the processor utilization on the Zabbix server and a significant decrease in the growth of data on its hard drive.

Klíčová slova

Monitorování počítačových sítí, Zabbix, PostgreSQL

Keywords

Computer network monitoring, Zabbix, PostgreSQL

Citace

Peter Drienko: Optimalizace dohledového nástroje pro správu sítě VUT, diplomová práce, Brno, FIT VUT v Brně, 2014

Optimalizace dohledového nástroje pro správu sítě VUT

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Tomáše Podermańského. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Peter Drienko
23. května 2014

Poděkování

Děkuji Ing. Tomáši Podermańskimu za odbornou pomoc a cenné připomínky při vedení práce.

© Peter Drienko, 2014.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Monitorovanie počítačových sietí **4**
2.1 Aktívne monitorovanie 4
2.2 Pasívne monitorovanie 5
2.2.1 Simple Network Management Protocol 6
2.2.2 Remote Monitoring 7
2.2.3 NetFlow 8
2.2.4 sFlow 9
2.2.5 Intelligent Platform Management Interface 10

3 Monitorovací nástroj Zabbix **12**
3.1 Komunikačný protokol 13
3.1.1 Pasívne kontroly 13
3.1.2 Aktívne kontroly 14
3.2 Tok informácií 16
3.3 Ukladanie dát 17
3.4 PostgreSQL 17
3.4.1 pgSQL 18
3.4.2 Partitioning v PostgreSQL 19
3.4.3 Plánovanie dotazov 22
3.5 Tabuľky databázy 23
3.5.1 Tabuľky history_* 24
3.5.2 Tabuľky trends_* a events 25
3.6 Veľkosť databázy 25
3.7 Housekeeper 26
3.7.1 Nevýhody funkcie housekeeper 27

4 Stav Zabbix serveru pred optimalizáciou **29**

5 Optimalizácia schématu databázy **34**
5.1 Riešenie vyžadujúce prázdnu databázu 34
5.1.1 Vytváranie partícií 34
5.1.2 Vytvorenie triggeru pre každú tabuľku 36
5.1.3 Vkladanie záznamov 36
5.1.4 Analýza riešenia 37
5.2 Riešenie nevyžadujúce prázdnu databázu 37
5.2.1 Rozdeľovacia funkcia 37

5.2.2 Vytvorenie triggeru pre každú tabuľku 39
5.2.3 Mazacia funkcia 39
5.2.4 Automatické mazanie partícií 40
5.2.5 Migrácia starých záznamov do partícií 40
5.2.6 Analýza riešenia 40
5.3 Výber optimalizovaných tabuliek 41
5.4 Úprava riešenia 41

6 Zhodnotenie výsledkov testovania 45

7 Záver 49

A Seznam použitých zkratek 53

B Obsah CD 54

Kapitola 1

Úvod

V súčasnosti je počítačová sieť považovaná za jednu z najdôležitejších zdrojov v organizácií, či už ide o privátny, alebo verejný sektor. Ak je časť siete neprístupná čo i len na krátku dobu, môže to výrazne ovplyvniť produktivitu organizácie, poškodiť jej aktíva a ohroziť jej dobré meno. Dôvodov pre zlyhanie siete môže byť niekoľko. Či už ide o škodlivé útoky, chyby používateľov, alebo výpadky zariadení, vždy existuje šanca, že používatelia zostanú bez prístupu k svojej službe. Pre zaistenie funkčnosti služieb kritických pre danú organizáciu je preto potrebné správať sa v tomto smere proaktívne a zabezpečiť monitorovanie sieťovej premávky.

Práve monitorovanie sieťovej premávky je hlavnou témou tejto diplomovej práce. Jej cieľom je vytvoriť ucelený dokument o problematike monitorovania počítačových sietí so zameraním sa na monitorovací nástroj Zabbix a dôrazom na prácu s veľkým objemom dát, ktoré sa pravidelne generujú vo VUT sieti.

Práca je rozdelená na dve časti. V prvej časti, ktorá bola vypracovaná ako semestrálny projekt, sa venujeme technológiám a problematike monitorovania počítačových sietí a popíšeme vlastnosti monitorovacieho nástroja Zabbix a možnosti jeho použitia.

V druhej časti vychádzame z poznatkov nadobudnutých o systéme Zabbix počas tvorby semestrálneho projektu, porovnávame existujúce riešenia optimalizácie schémy databázy, najvhodnejšie z nich implementujeme a na základe toho navrhujeme nové, optimálnejšie riešenie. Túto optimalizáciu otestujeme v sieti VUT, zhodnotíme výsledky riešenia a popíšeme krátkym inštalačným návodom pre sieťovú komunitu.

Kapitola 2

Monitorovanie počítačových sietí

Monitorovanie je jednou z hlavných aktivít správy počítačových sietí. Správa počítačovích sietí je definovaná ako aktivity, metódy, procedúry a nástroje, ktoré sa týkajú prevádzky, administrácie, údržby a zabezpečenia služieb počítačových systémov.

Prevádzka sa zaoberá udržiavaním siete (a služieb, ktoré sieť poskytuje) online a jej bezproblémovým chodom. Prevádzka zahŕňa aj monitorovanie siete, aby bolo možné zistiť problém tak rýchlo ako je možné, ideálne pred tým, ako je ovplyvnená funkcionalita u používateľov. Administrácia zahŕňa sledovanie zdrojov v sieti a a tým ako sú prideľované. Pri údržbe ide najmä o opravy a aktualizácie - napríklad keď smerovač potrebuje nový operačný systém, alebo keď sa pridáva nový prepínač do siete. Údržba tiež zahŕňa korekciu a preventívne opatrenia ako napríklad úprava parametrov zariadení tak, ako je to potrebné. Zabezpečenie služieb je spojené s konfiguráciou zdrojov v sieti tak, aby bolo možné dané služby podporovať. To môže znamenať napríklad nastavenie siete tak, aby nový používateľ dokázal používať hlasové služby [1].

Táto práca sa zameriava najmä na prevádzku siete a jej monitorovanie, v ďalších kapitolách sa teda ostatným aktivitám správy sietí venovať nebudeme.

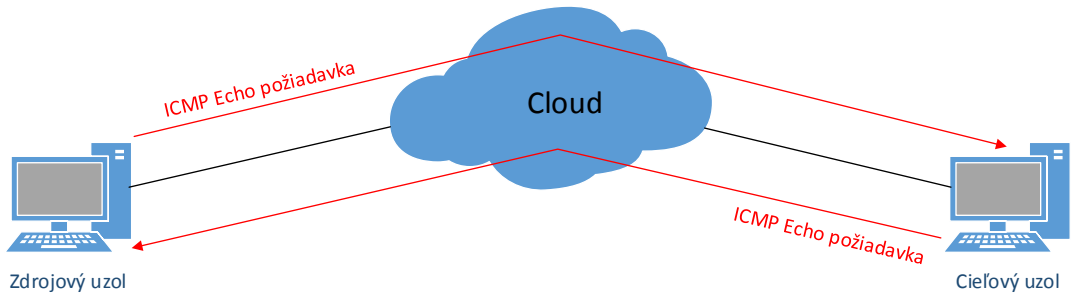
Vo všeobecnosti môžeme podľa [2] rozdeliť základné metodiky monitorovania sietí na dva typy: aktívne a pasívne. V nasledujúcej časti tieto metodiky popíšeme a zameriame sa hlavne na pasívne monitorovanie sietí.

2.1 Aktívne monitorovanie

Pri aktívnom monitorovaní sa zvyčajne do siete vkladajú pakety a následne pozorujú rozličné sieťové metriky. Sledujú sa metriky ako: dostupnosť, oneskorenie paketov, strata paketov, šírka pásma, alebo priepustnosť. Na meranie týchto metrík sú často použité nástroje ako ping, ktorý merá stratu a omeškanie paketov, alebo traceroute, ktorý pomáha pri určovaní trasy k cieľovému uzlu. Oba nástroje vkladajú do siete ICMP pakety (obrázok 2.1), ktoré sú doručené cieľovému uzlu a čakajú na jeho odpoveď.

Problém aktívneho monitorovania môže byť rušenie bežnej premávky s tou, ktorá je do premávky vkladaná pri aktívnom monitorovaní a môže tiež zvýšiť zaťaženie siete. S monitorovacími paketmi sa často zaobchádza odlišne ako s bežnou premávkou, môže to teda viesť k nesprávnym údajom. Z týchto dôvodov sa aktívne monitorovanie len zriedkavo používa samostatne.

Na druhej strane aktívny prístup poskytuje explicitnú kontrolu nad generovaním paketov. To zahŕňa kontrolu nad pôvodom sieťovej premávky, typom vzorkovacích techník,

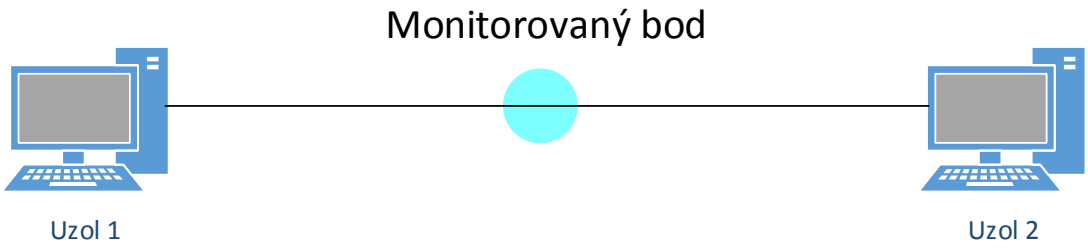


Obrázek 2.1: Aktívne monitorovanie pomocou ICMP [3]

časovaním, frekvenciou, plánovaním, veľkosťami a typmi paketov (pre emulovanie rozličných aplikácií), alebo cestou a funkciou, ktorú sa rozhodneme monitorovať. Byť aktívny implikuje testovanie toho čo chceme a kedy to potrebujeme [4].

2.2 Pasívne monitorovanie

Na rozdiel od aktívneho monitorovania, pri pasívnom sa do siete nevkladajú žiadne pakety. Pasívne monitorovanie neintruzívne monitoruje jeden bod v sieti (obrázok 2.2), zatiaľčo aktívne poskytuje informácie o ceste medzi dvomi uzľami. Pri pasívnom monitorovaní môžeme dáta zbierať buď z liniek použitím sondy (napr. tcpdump) na sledovanie prechádzajúcej sieťovej premávky, alebo z pripojených uzlov (obvykle smerovače, prepínače, alebo servery).



Obrázek 2.2: Monitorovaný bod na sledovanej linke [3]

Pasívne monitorovanie pozoruje metriky ako miera použitia protokolov, správanie používateľov, alebo miera chybovosti paketov. Pasívne monitorovanie síce tak nezaťažuje sieť ako aktívne, pri pasívnom monitorovaní však môžeme sieť analyzovať až po zozbieraní údajov, nie priamo pri ich zbere. Z toho vyplýva problém pri spracovaní veľkého objemu uložených dát.

Keďže majú aktívne a pasívne monitorovanie svoje výhody aj nevýhody, často sa používa ich kombinácia [4].

Pri pasívnom monitorovaní sa najviac používajú protokoly SNMP, NetFlow, sFlow, RMON a IMPI, ktoré si v nasledujúcej časti popíšeme.

2.2.1 Simple Network Management Protocol

Simple Network Management Protocol (SNMP) je protokol pre správu zariadení v IP sieťach. Používa sa najmä na monitorovanie stavu zariadení (napr. teplota vo vnútri prepínaču) a nastavovanie zariadení (vypnutie Ethernetového rozhrania).

SNMP rozlišuje dve základné entity: agent a manager. Manager je často označovaný ako Network Management Station (NMS). NMS je zodpovedný za prijímanie a žiadanie trap správ z agentov v sieti. Žiadanie v kontexte sieťovej správy znamená požiadať agenta o nejaký typ informácie, ktorý je neskôr použitý na zistenie, či nastala určitá udalosť, ktorú je potrebné vyriešiť.

Trap je asynchrónne zasielaná správa a NMS je zodpovedný za vykonanie akcie podľa informácie, ktorú príjme v tejto trap správe od agenta. Ako príklad môžeme uviesť zlyhanie Ethernetového portu smerovača, čo spôsobí, že smerovač zašle trap správu do NMS, ktoré môže administrátora následne upozorniť o zlyhaní.

Agent je softvér implementovaný v sieťových zariadeniach, ktoré administrátor spravuje. Môže ísť o samostatný program (v UNIX systémoch daemon), alebo môže byť začlenený do operačného systému (súčasť IOS v Cisco zariadeniach). V súčasnosti je SNMP agent zabudovaný vo väčšine IP zariadeniach. Agent poskytuje rozličné informácie o stave zariadení pre NMS. Napríklad agent na smerovači dokáže sledovať jeho rozhrania a zaslať trap správu o ich stave NMS. Obrázok 2.3 zobrazuje vzťah medzi NMS a Agentom.

Okrem zasielania trap správ od agenta k NMS je možný aj druhý spôsob fungovania SNMP, ktorý sa volá polling. NMS pri tomto spôsobe aktívne žiada o dáta. Na obrázku 2.3 vidíme, ako polling pracuje. NMS vyšle požiadavku k agentovi a agent na ňu odpovedá.

Stav zariadení, alebo štatistické informácie ku ktorým má NMS prístup sú definované v Management Information Base (MIB). MIB definuje vždy názov objektu a následne je vysvetlený jeho význam.

Obrázek 2.3: Vzťah medzi NMS a Agentom [3]

SNMP používa UDP ako transportný protokol na zasielanie datagramov medzi NMS a agentom. Nie je teda vytvárané spojenie, čo znamená, že datagramy nie sú potvrdzované na úrovni protokolu. Je teda na aplikačnej vrstve, či zabezpečí detekciu strát datagramov a znovu ich prepošle. Výhodou použitia UDP je však nízka réžia, takže nadmerne nezaťažuje sieť.

Keď SNMP pracuje v režime polling, problém straty datagramov je väčšinou riešený jednoduchým vypršaním časového limitu. Problém však nastáva pri zasielaní trap správ od Agenta kedy NMS nemá ako vedieť, že trap správa bola vôbec zaslaná. Agent zase nevie,

že musí dáta poslať znovu, keďže NMS nie je povinný zaslať odpoveď potvrdzujúcu prijatie trap správy. Najmä preto sa v praxi využíva najmä SNMP polling.

V súčasnosti má SNMP tri verzie: SNMPv1, SNMPv2 a SNMPv3.

SNMPv1 je prvá verzia a je to v podstate historický IETF štandard. Bezpečnosť v SNMPv1 je založená na komunitách, ktoré predstavujú autentizáciu pomocou hesla zasielaného bez akéhokoľvek šifrovania. Prvá verzia SNMP protokolu používa 32 bitové počítadlá, čo nie je postačujúce pre sledovanie šírky pásma spojenia na úrovni Gigabitov za sekundu. Síce už ide o historický štandard, stále je veľa výrobcov sieťových zariadení, ktorí ho podporujú.

SNMPv2 už podporuje 64 bitové počitadlá, pridáva niekoľko nových typov datagramov, avšak heslá sú stále prenášané nešifrované. Výrazný posun nastal tiež v prenášaní dát. Pri SNMPv1 sa na prenos jednej metriky použil jeden datagram, pri SNMPv2 je to efektívnejšie vďaka novej operácií bulkwalk, ktorá umožňuje zaslanie viacerých metrík v jednom UDP datagrame.

SNMPv3 už nepoužíva komunity ako predchádzajúce verzie. Miesto komunít umožňuje nastaviť pri autentifikácií používateľské účty s heslami a taktiež umožňuje šifrovanie datagramov. Je však o niečo zložitejší na nastavenie a prináša vyššiu réžiu, keďže sú hlavičky datagramov dlhšie a komplexnejšie a šifrovacie výpočty môžu zvýšiť záťaž CPU až o 20%. SNMPv3 sa aj pre tieto dôvody v praxi používa len málo [5].

2.2.2 Remote Monitoring

Remote Monitoring (RMON) je jednou z mnohých MIB patriacich SNMP. Bol pôvodne určený na správu vzdialených sietí z centrálneho miesta. RMON používa monitorovacie zariadenia nazývané sondy (probes), ktoré zbierajú informácie o sieti a analyzujú pakety. Sondy dokážu analyzovať len tú sieťovú premávku, ktorá cez nich prechádza, preto musia byť umiestnené na každom segmente siete, ktorý má byť monitorovaný. Pôvodne mali byť sondy samostatné zariadenia, v súčasnosti sú však skôr súčasťou iných sieťových zariadení.

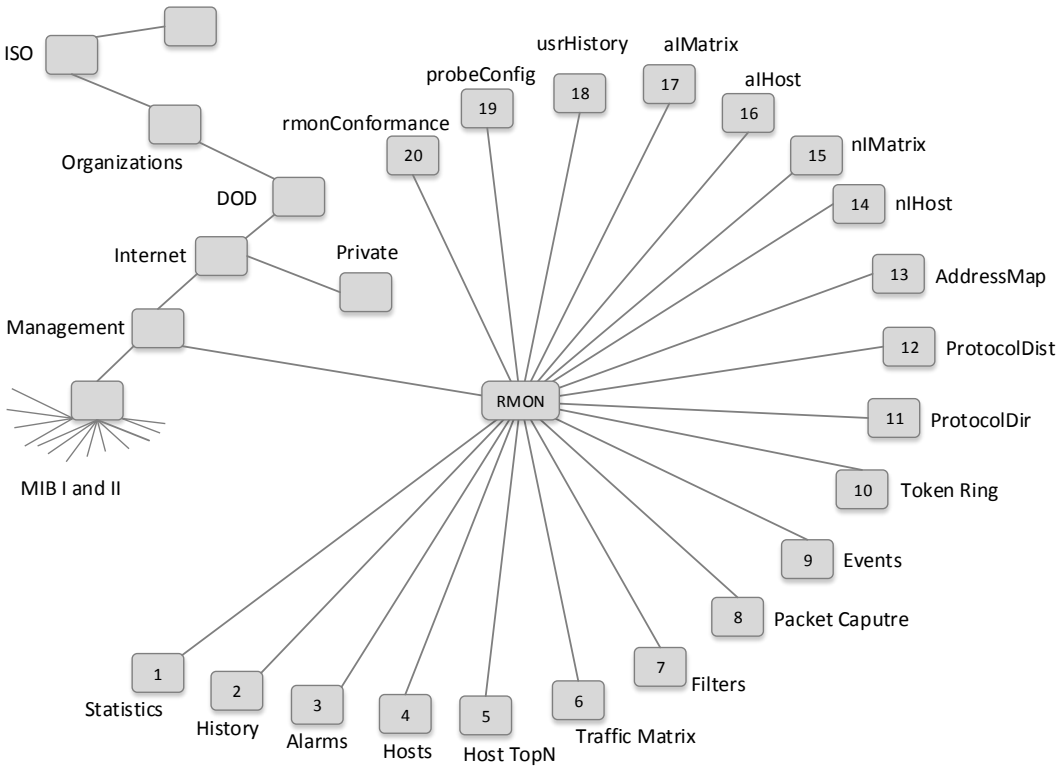
Druhý prvok RMON je správcovská stanica (Management Station), ktorá od sond prostredníctvom SNMP získava a vyhodnocuje RMON dáta. RMON je navrhnutý tak, aby operoval odlišne od iných systémov založených na SNMP:

- sondy majú viac zodpovednosti na zbere a spracovaní dát, čo znižuje SNMP sieťovú premávku a výkonnostné zaťaženie na klientoch
- informácie sú miesto nepretržitého dotazovania prenášané k spravujúcej aplikácií len keď je to požadované

RMON je viac podobný technológiám založeným na toku dát ako napríklad NetFlow či sFlow, keďže zbierané dáta sú zamerané skôr na vzory sieťovej premávky ako na stav individuálnych zariadení. Nevýhoda takéhoto systému je, že vzdialené zariadenia sú viac zaťažené správou a potrebujú na to viac zdrojov.

Niektoré zariadenia preto implementujú len časť RMON MIB skupín. Minimálna implementácia RMON agenta by mala obsahovať štatistické, historické, alarmové a udalostné MIB (skupiny 1,2,3 a 9 na obrázku 2.4) skupiny.

Pôvodný RMON označovaný ako RMON 1 zahŕňa iba linkovú vrstvu RM OSI modelu (skupiny 1 až 10 na obrázku 2.4). RMON 2 potom dopĺňa možnosť monitorovania od sieťovej až po linkovú vrstvu (skupiny 11 až 20). Verzie 1 a 2 poskytujú len 32 bitové počítadlá, pre použitie 64 bitových je potrebné siahnuť po novšom štandarde HC-RMON (RFC 3273), alebo RMON2 MIB (RFC4502) [6].



Obrázek 2.4: RMON MIB strom

2.2.3 NetFlow

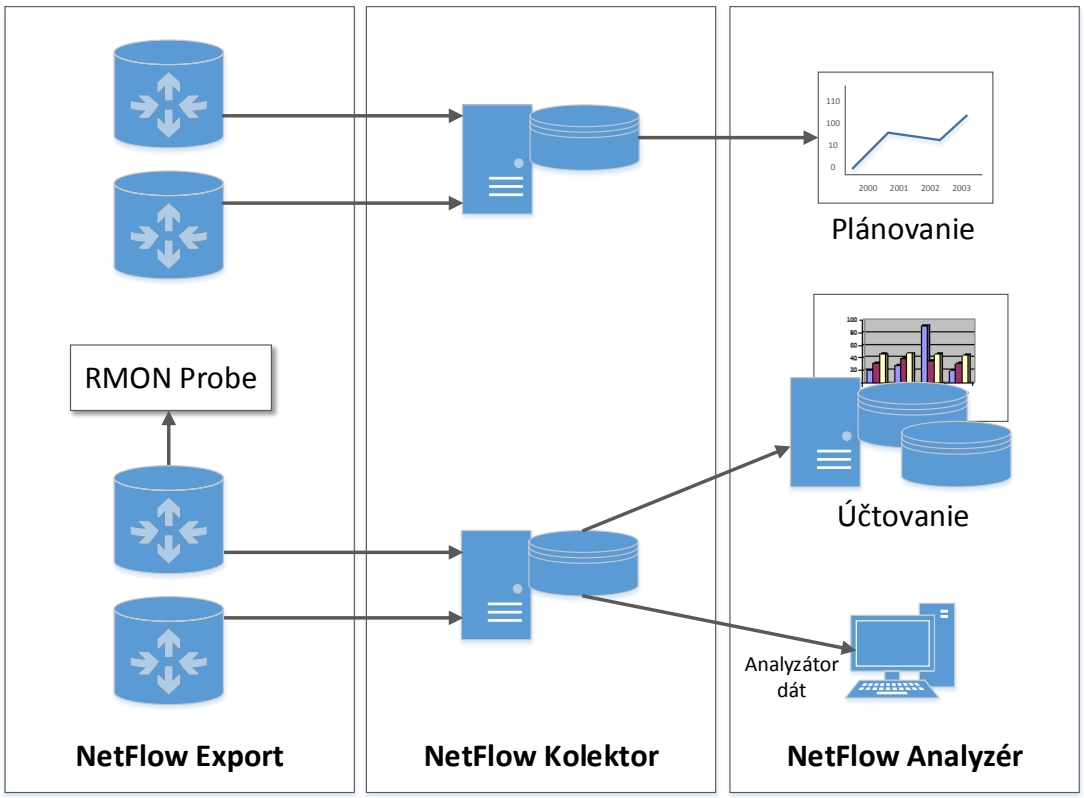
NetFlow je sieťový protokol vyvinutý spoločnosťou Cisco na zber informácií o sieťovej premávke, ktorá prechádza sieťovým rozhraním daného zariadenia. NetFlow bola pôvodne na krátku dobu softvérová technológia od spoločnosti Cisco používaná v IOS 11.x na prepínanie paketov pre Cisco smerovače rady 7000, 7200 a 7500. NetFlow sa neskôr ukázal byť nevhodný pre veľké smerovače, najmä pre internetové, chrbticové smerovče, kde je počet súbežných tokov oveľa dôležitejší ako ten v lokálnych sieťach a kde niektoré typy premávky spôsobujú veľké množstvo krátko trvajúcich tokov, ako napríklad DNS požiadavky. Preto bol NetFlow ako prepínacia technológia nahradený technológiou Cisco Express Forwarding [7].

NetFlow tok môže byť definovaný viacerými spôsobmi. Cisco štandard verzie 9 [8] definuje tok ako jednosmerná sekvencia paketov zdieľajúca nasledujúcich 7 hodnôt:

- vstupné rozhranie
- zdrojová IP adresa
- cieľová IP adresa
- IP protokol
- zdrojový port pre UDP a TCP, 0 pre ostatné protokoly
- cieľový port pre UDP a TCP, typ a kód pre ICMP, 0 pre ostatné protokoly

- typ služby IP protokolu

Analýzou dát získaných z NetFlow dokáže administrátor určiť informácie ako zdrojový a cieľový uzol sieťovej premávky, triedu služby (CoS), alebo dôvod zahltenia. Typickým príkladom použitia NetFlow môže byť sledovanie aký typ premávky vchádza, alebo vychádza do korporátnej siete, alebo sledovanie tokov medzi BGP autonómnymi systémami. Na obrázku 2.5 vidíme základné prvky protokolu NetFlow.



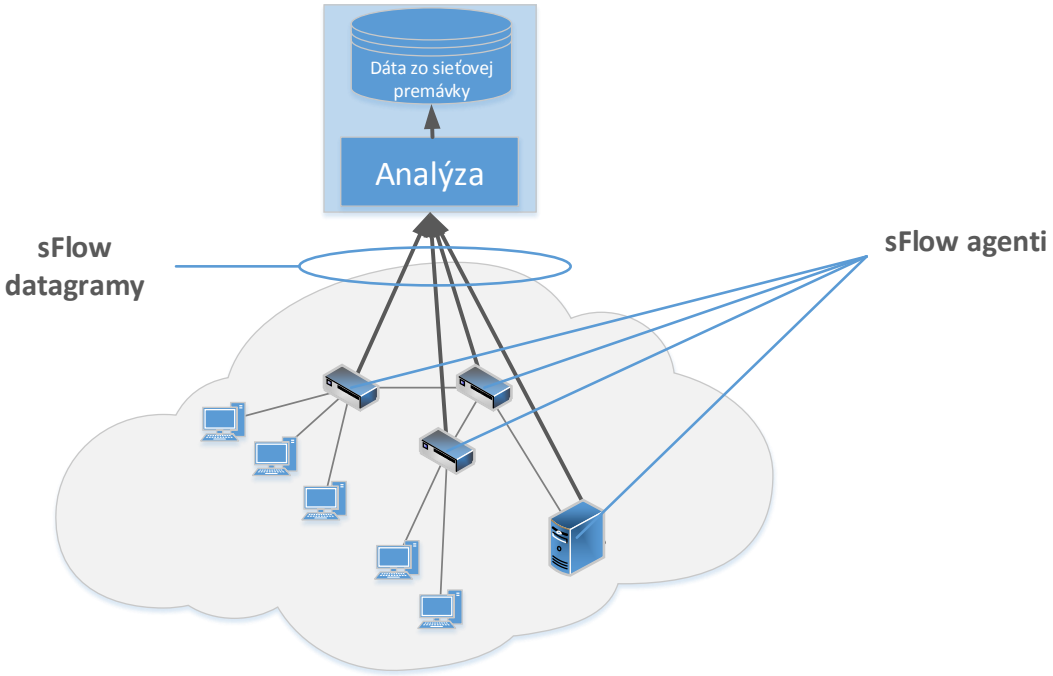
Obrázek 2.5: Infraštruktúra v NetFlow

NetFlow Export analyzuje a zbiera dátové toky vstupujúce do rozhrania prepínača, alebo smerovača a pripravuje ich na export. NetFlow Kolektor zachytáva exportované dáta z viacerých smerovačov a následne ich filtruje a agreguje podľa nastavených politík a následne tieto agregované dáta ukladá. NetFlow Analyzér zobrazuje a aktualizuje agregované dáta vo vopred nastavenom časovom interval [9].

2.2.4 sFlow

sFlow je vzorkovacia technológia zabudovaná do prepínačov a smerovačov. Poskytuje schopnosť nepretržite monitorovať sieťovú premávku, súbežne na všetkých sieťových rozhraniach. Obrázok 2.6 zobrazuje základné prvky sFlow systému. sFlow agenti nepretržite rozosielajú prúd sFlow datagramov do centrálneho sFlow Kolektoru, kde sú analyzované a následne poskytujú bohaté informácie o sieťovej premávke v reálnom čase.

Monitorovanie rýchlych, smerovaných a prepínaných sietí má nasledujúce vlastnosti:



Obrázek 2.6: Základné prvky sFlow

- Detailnosť: Kompletná hlavička paketu a informácie o prepínaní/smerovaní umožňujú detailnú analýzu od druhej až po siedmu vrstvu RM OSI
- Škálovateľnosť: sFlow systém je škálovateľný vo veľkosti a rýchlosti siete, ktorú dokáže monitorovať. sFlow dokáže monitorovať siete s rýchlosťou prenosu dát 100 Gbps a viac. Tisíce zariadení môže byť monitorovaných jediným sFlow kolektorom
- Nízke náklady: sFlow agent je veľmi jednoduchý na implementáciu, pridáva tak len zanedbateľné zvýšenie výrobných nákladov
- Aktuálny: sFlow kolektor má vždy aktuálny pohľad na premávku v celej sieti. Aktuálne informácie sú špeciálne dôležité ak je potrebné poskytovať kontrolu nad premávkou v reálnom čase (napr. pri obrane proti DoS útokom, alebo pri nasadení kvality služby)

sFlow sa v porovnaní s NetFlow používa najmä v heterogénnom prostredí s viacerými protokolmi, zatiaľčo NetFlow sa používa v sieťach podporujúcich len protokol IP. Z dôvodu, že sFlow je vzorkovacia technológia, ktorá zachytáva len vzorky a nie všetky dáta je v prípade, že potrebujeme 100% presnosť o sieťovej premávke vhodnejšie použiť NetFlow [10].

2.2.5 Intelligent Platform Management Interface

Intelligent Platform Management Interface (IPMI) je štandard, ktorý bol predstavený v roku 1998 spoločnosťami Dell, HP, Intel a NEC. IPMI definuje systémové a sieťové rozhrania používané na prístup, monitorovanie, diagnostiku a obnovu počítačových systémov.

Tieto funkcie sú sprístupnené špeciálnym procesorom s názvom Baseboard Management Controller (BMC), ktorý je umiestnený na základnej doske serveru. Procesor BMC je s hlavným procesorom a ďalšími jednotkami základnej dosky spojený jednoduchou sériovou zbernicou. Pomocou nej monitoruje parametre základnej dosky (teplota, status CPU, otáčky ventilátoru, napájanie), umožňuje vzdialené riadenie napájania serveru (schopnosť reštartu serveru), prístup ku konfigurácií BIOS a informáciám konzoly operačného systému. Keďže je BMC samostatný procesor, systém pracuje bez ohľadu na to, či hlavný procesor beží, alebo nie. Verzia 1 používala na komunikáciu sériové rozhranie, no od verzie 1.5 je použité LAN rozhranie. Verzia 2.0, čo je v súčasnosti najpoužívanejšia verzia zahŕňa základné bezpečnostné funkcie, VLAN a blade podporu.

Ďalšou kľúčovou funkcionalitou IPMI je Serial Over LAN (SOL). IPMI komunikácia je založená na IP protokole, no SOL funkcia poskytuje vzdialené pripojenie k systému prostredníctvom sériovej konzoly. SOL presmerováva lokálne sériové rozhranie cez UDP reláciu čo umožňuje vzdialený prístup k sériovej konzole servera. To je dosiahnuté IPMI firmvérom, ktorý zachytáva dáta z COM portu a následne ich preposiela cez LAN sieť. Vďaka tejto funkcii je možné vzdialene nahliadať do BOOT, alebo núdzovej konzoly a umožňuje diagnostikovať a opraviť problémy na serveri.

Napriek spomínaným vlastnostiam nie je používanie štandardu IPMI v podnikových sieťach veľmi rozšírené [11].

Kapitola 3

Monitorovací nástroj Zabbix

Zabbix je semi-distribuovaný, voľne šíriteľný, monitorovací systém s centrálnym riadením. Nástroj Zabbix ponúka nasledujúcu funkcionalitu:

- centralizované, ľahko použiteľné webové rozhranie
- server, ktorý funguje na väčšine operačných systémoch založených na Unixe ako napríklad Linux, AIX, FreeBSD, OpenBSD, alebo Solaris
- agenti, ktorí fungujú na operačných systémoch založených na Unix a Windows
- schopnosť priamo monitorovať zariadenia s SNMP (v1, v2 a v3) a IPMI funkcionalitou
- schopnosť analyzovať zhromaždené dáta pomocou grafov a iných vizuálnych techník
- notifikácie, ktoré umožňujú ľahkú integráciu s ostatnými systémami
- flexibilnú konfiguráciu s možnosťou vytvárať vzory
- schopnosť automaticky objaviť nové zariadenia v sieti

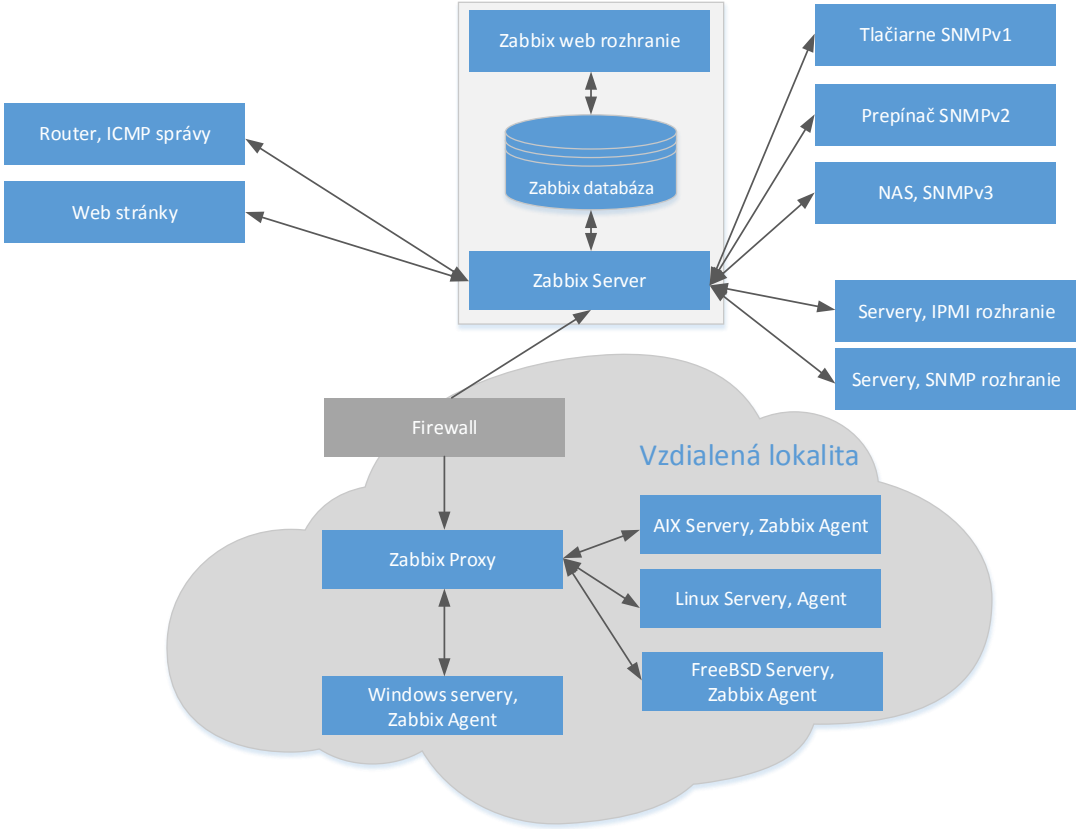
Na obrázku 3.1 môžeme vidieť zjednodušenú sieť možného zapojenia systému Zabbix znázorňujúcu niekoľko možných monitorovacích schopností a zariadení.

Centrálnym prvkom je databáza, ktorá môže byť implementovaná ako PostgreSQL, MySQL, Oracle a iné databázové riešenia. Zabbix server, ktorý je napísaný v programovacom jazyku C a Zabbix webové rozhranie napísané v PHP môžu byť na jednom fyzickom stroji, alebo každý samostatne na svojom stroji. Samostatne môžu byť prvky rozložené napríklad v prípade, že by sme chceli využiť vysoko výkonný databázový, alebo webový server. Keď každý prvok beží na samostatnom stroji, server a webové rozhranie potrebujú prístup k databáze a webové rozhranie môže potrebovať prístup k serveru, aby mohol zobraziť jeho status.

Zabbix server priamo monitoruje viacero zariadení, avšak pokiaľ je zariadenie vo vzdialenej lokalite za firewallom, dáta získava prostredníctvom Zabbix proxy. Zabbix proxy a agenti sú oba napísané v jazyku C.

Monitorované zariadenia majú len malú kontrolu nad tým čo je monitorované, keďže je väčšina konfigurácie riadená centralizovane. Tento prístup významne znižuje možnosť, že jedno zariadenie naruší konfiguráciu celej siete [12].

Prvky nástroja Zabbix komunikujú prostredníctvom vlastného agent-server protokolu, ktorý si popíšeme v nasledujúcej kapitole.



Obrázek 3.1: Jednoduchý model zapojenia systému Zabbix

3.1 Komunikačný protokol

Komunikačný protokol nástroja Zabbix je založený na štandarde JSON. Protokol rozlišuje dva spôsoby komunikácie:

- pasívne kontroly
- aktívne kontroly

3.1.1 Pasívne kontroly

Pasívne kontroly sú jednoduché požiadavky serveru (alebo proxy) k agentovi na zaslanie dát ako napríklad zaťaženie CPU, na ktorú agent odpovedá s aktuálnymi hodnotami. Formát požiadavky od servera je nasledovný:

```
<item key>\n
```

Agenta odpovie správou:

```
<HEADER><DATALEN><DATA>
```

Príklad komunikácie môže byť nasledovný [13]:

1. Server vytvorí TCP spojenie
2. Server zašle agentovi správu: agent.ping\
3. Agent odpovie správou: <HEADER><DATALEN>1
4. Server spracuje prijatú hodnotu '1', čo v tomto prípade znamená, že daný agent je dostupný
5. TCP spojenie sa ukončí

3.1.2 Aktívne kontroly

Aktívne kontroly vyžadujú o niečo komplexnejšie spracovávanie ako pasívne. Agent musí najskôr získať od servera zoznam metrík, ktoré má sledovať a následne ich posiela v periodických intervaloch. Získavanie zoznamu metrík prebieha tak, že agent zašle požiadavku:

```
<HEADER><DATALEN>{
  "request":"active checks",
  "host":"<hostname>"
}
```

Následne server odpovie so správou:

```
{
   "response":"success",
   "data":[
   {
      "key":"log[\/home\/zabbix\/logs\/zabbix_agentd.log]",
      "delay":"30",
      "lastlogsize":"0"
   },
   {
      "key":"agent.version",
      "delay":"600"
   }
   ]
}
```

Server musí odpovedať so "success". Pre každú metriku je potrebný kľúč s názvom danej metriky a frekvencia, popisujúca ako často bude agent zasielať aktualizované údaje. Pri type metriky "log"je potrebné zaslať aj údaj o poslednej veľkosti súboru. Príklad komunikácie pri získavaní metrík môže byť nasledovný:

1. Agent vytvorí TCP spojenie
2. Agent požiada server o zoznam kontrolovaných metrík

3. Server odpovie so zoznamom metrík (názov metriky, frekvencia aktualizácií)
4. Agent spracuje odpoveď
5. TCP spojenie sa ukončí
6. Agent začne periodicky posielať zmerané hodnoty metrík

Zasielanie nameraných hodnôt od agenta prebieha nasledovne:

```
<HEADER><DATALEN>{
   "request":"agent data",
   "data":[
       {
           "host":"<hostname>",
           "key":"log[\/home\/zabbix\/logs\/zabbix_agentd.log]",
           "value":" 13039:20090907:184546.759 zabbix_agentd started. ZABBIX 1.6.6
           (revision {7836}).",
           "lastlogsize":80,
           "clock":1252926015
       },
       {
           "host":"<hostname>",
           "key":"agent.version",
           "value":"1.6.6",
           "clock":1252926015
       }
   ],
   "clock":1252926016
}
```

Odpoveď servera:

```
<HEADER><DATALEN>{
    "response":"success",
    "info":"Processed 2 Failed 0 Total 2 Seconds spent 0.002070"
}
```

Server vždy potvrdzuje prijatie hodnôt a čas trvania odpovede. Príklad komunikácie pri zasielaní dát môže byť nasledovný [13]:

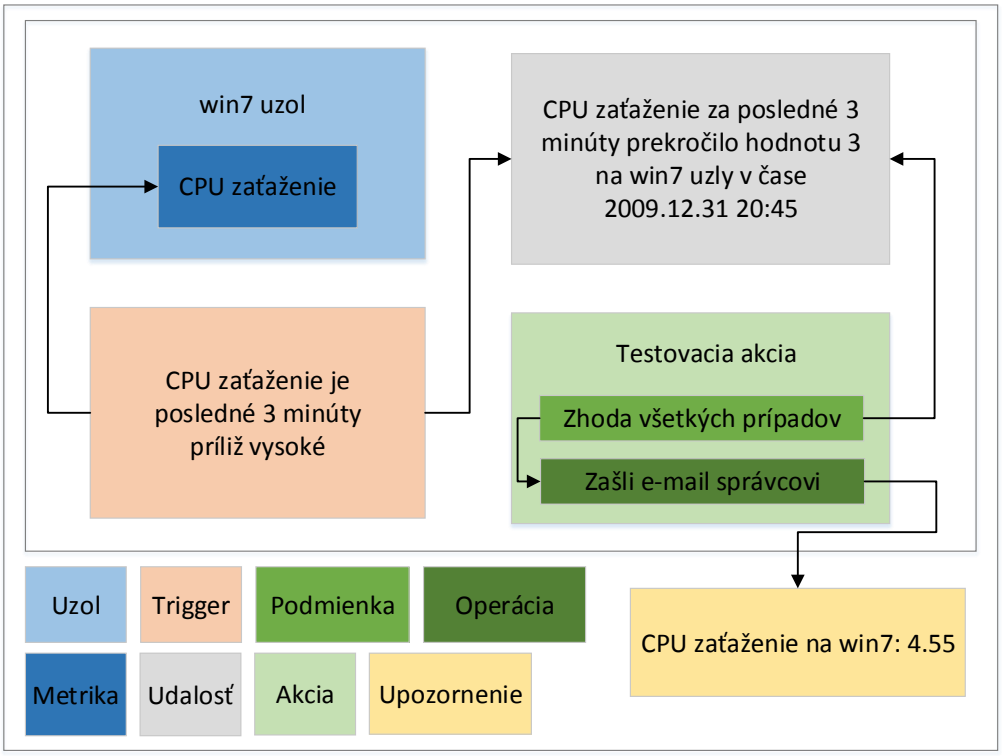
1. Agent vytvorí TCP spojenie
2. Agent zašle zoznam nameraných hodnôt
3. Server príjme zmerané hodnoty

4. Server pošle potvrdenie o prijatí Agentovi
5. TCP spojenie sa ukončí

3.2 Tok informácií

Pre lepšie pochopenie princípov fungovania systému Zabbix popíšeme v tejto kapitole ako Zabbix server vyhodnocuje údaje a vykonáva následné opatrenia. Na obrázku 3.2 vidíme jednotlivé entity Zabbix servera, ktoré sa zúčastňujú základného toku informácií.

Celý tok informácií začína u agenta, ktorý sa v Zabbix serveri nazýva *uzol* (host). Každý uzol obsahuje zoznam *metrík* (items), ktoré sú sledované *triggerom*. Kedykoľvek trigger zistí zhodu v nastavenom výraze, zmení svoj stav na PROBLEM. Keď už zhoda vo výraze nie je platná, zmení stav na OK. Vždy keď nastane zmena v stave triggeru, vygeneruje sa nová *udalosť* (event). Udalosť obsahuje podrobné informácie o zmene triggeru - kedy nastala zmena a aký je nový stav. Udalosť môže byť okrem triggeru generovaná aj pri objavení nových uzlov, alebo služieb. Na udalosti reagujú *akcie*, ktoré je možné nastaviť vo webovom rozhraní. Pri vytváraní akcie môžeme nastaviť *podmienky*, podľa ktorých sa bude reagovať na rôzne udalosti. Na obrázku nie je nastavená podmienka, preto sa reaguje na všetky udalosti. Každá akcia tiež obsahuje operáciu, ktoré definujú čo sa má presne stať ak je podmienka akcie splnená. Na konci sa vygeneruje *upozornenie*, ktoré je zvyčajne vykonávané mimo Zabbix server, ako napríklad zaslanie e-mailu.



Obrázek 3.2: Nastavenie typu metriky

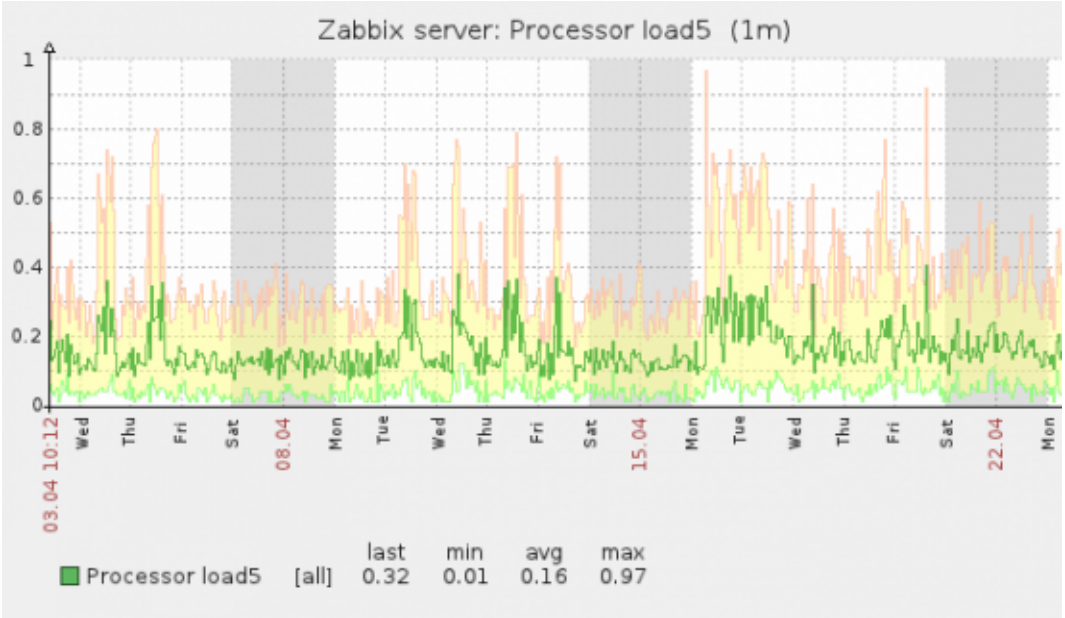
Trigger môže byť taktiež v stave UNKNOWN. Toto nastáva keď bolo nastavenie triggeru

upravené a ten ešte nemá dostatok dát aby určil súčasný stav. Udalosti pre zmenu do a zo stavu UNKNOWN sa pri podmienke v akcií nekontrolujú. Všetky spomínané operácie s výnimkou upozornenia vykonáva centrálne proces Zabbix server [12].

3.3 Ukladanie dát

Archivácia dát získaných z monitorovania sieťovej premávky môže mať niekoľko dôvodov. Či už ide o vykreslenie grafu monitorovaných metrík v čase, spätnú analýzu útokov na sieť, alebo politiku organizácie nariaďujúcu uchovávať údaje po určitú dobu, archivácia dát je pri monitorovaní sietí dôležitá.

Server systému Zabbix ukladá monitorované údaje do databázy, z ktorej je následne možné vyhodnocovať dáta napríklad prostredníctvom grafov vo webovom rozhraní, čo vidíme na obrázku 3.3. Tento graf zobrazuje zhromaždené údaje o zaťažení procesoru zariadenia. Tieto údaje boli uložené do tabuľky trends, ktorú si v tejto kapitole popíšeme a následne vykreslené v grafe. Používateľ môže na tomto grafe vidieť maximálne (ružová), priemerné (tmavo-zelená) a minimálne (bledo-zelená) hodnoty pre daný deň [13].



Obrázek 3.3: Graf zaťaženia procesoru vo webovom rozhraní Zabbix [13]

3.4 PostgreSQL

Zabbix umožňuje používateľom vybrať si z niekoľkých databázových systémov. Keďže je v prostredí VUT siete v súčasnosti implementovaná verzia s PostgreSQL, budeme sa v práci zameriavať len na tento databázový systém.

PostgreSQL je voľne šíriteľný, objektovo relačný, databázový systém, ktorý bol vytvorený na Kalifornskej univerzite a prvý krát vydaný v roku 1995. PostgreSQL je udržiavaný komunitov vývojárov po celom svete, ktorí už stihli vydať 9 hlavných verzií a posledná verzie nesie označenie 9.2.4, ktorá je nasadená aj v sieti VUT. Z PostgreSQL je dnes podľa

[14] 4. najpoužívanejší databázový systém hneď po Oracle, MySQL a Microsoft SQL Server systémoch.

PostgreSQL obsahuje funkcionalitu, ktorú len ťažko nájdeme v iných voľne šíriteľných, databázových systémoch. Významnou funkcionalitou je možnosť používateľov napísať si vlastné procedúry a funkcie, ktoré budú vykonávané databázovým serverom v jednom z podporovaných programovacích jazykov. V súčastnosti podporuje pgSQL, Perl, Python, Java, Tcl, JavaScript, PHP a mnoho iných jazykov. PostgreSQL tiež umožňuje napísať funkcie v jazyku C, ktoré môžu byť neskôr zavolané rovnako ako hociktoré iné funkcie. Umožňuje tiež vytvárať funkcie v rôznych jazykoch, z ktorých sa následne vytvorí jeden dotaz na databázu, bez potreby kompilovania týchto funkcií, čo nie je napríklad v MySQL/SQL Server možné.

Dôležitou funkcionalitou je tiež schopnosť definovať si nové dátové typy, ktoré môžu byť použité ako stĺpce tabuliek, alebo nové operátory, funkcie či indexy. Je možné si dokonca vytvoriť stĺpec ako pole prvkov, používateľ si teda dokáže definovať napríklad stĺpec pre telefónne číslo, ktoré môže následne obsahovať niekoľko čísiel (pracovné číslo, číslo domov, atď).

PostgreSQL umožňuje používateľom inštalovať mnohé rozšírenia, ktoré obohacujú funkcionalitu tohto systému. Ide napríklad o rozšírenie PostGIS, ktoré umožňuje použiť PostgreSQL ako databázu pre geografické informačné systémy a následne povoľuje vykonávať dotazy založené na polohe typu: "nájdi N najbližších bodov od tejto polohy". PostPIC rozšírenie zase pridáva možnosť spracovávať obrázky vo vnútri databázy takže si používateľ dokáže orezať, zmeniť veľkosť, alebo získať atribúty jednotlivých obrázkov. V základe PostgreSQL neumožňuje vykonávať dotazy nad viacerými databázami súčasne, to však rieši rozšírenie DBLINK, ktoré túto funkcionalitu pridáva. Od verzie 9.1 je inštalovanie týchto rozšírení tak jednoduché, ako spustenie jedného SQL dotazu typu CREATE EXTENSION [názov_rozširenia].

PostgreSQL chráni dáta a koordinuje viacero súbežných používateľov prostredníctvom transakčného modelu, ktorý využíva viac-verziové súbežné riadenie (MVCC). MVCC poskytuje oveľa lepšiu výkonnosť ako poskytujú ostatné produkty koordinujúce prístup viacerých používateľov cez zamknutie tabuľky, riadku, či stránky. PostgreSQL sa práve preto odporúča používať v prostredí s vysokým objemom dát a počtom transakcií [15].

Všetky spomínané funkcie, vysoká dostupnosť (málo výpadkov), integrácia s mnohými monitorovacími nástrojmi, vysoká bezpečnosť (SSL protokol pre TCP/IP spojenia, kryptografické funkcie pgcrypto, skupinové a používateľské oprávnenia) robia z PostgreSQL v súčasnosti najpokročilejší, voľne šíriteľný, databázový systém, často odporúčaný pre Zabbix servery monitorujúce rozsiahle siete.

PostgreSQL databáza v Zabbixe obsahuje vyše 100 tabuliek podľa typu dát, ktoré sa ukladajú, preto si najdôležitejšie z nich popíšeme v kapitole 3.5.

3.4.1 pgSQL

pgSQL je procedurálny jazyk pre PostgreSQL databázový systém. Ciele pri návrhu pgSQL boli vytvoriť procedurálny jazyk, ktorý:

- môže byť použitý na vytváranie funkcií a triggerov
- pridáva kontrolné štruktúry typu if-then do SQL jazyka
- dokáže vykonávať komplexné výpočty

- dedí používateľom definované typy, funkcie a operátori
- je ľahko použiteľný

SQL je jazyk, ktorý PostgreSQL a väčšina ostatných relačných databázových systémov používa ako dotazový jazyk. Je prenosný medzi rôznymi systémami a ľahko naučiteľný. Má však nevýhodu v tom, že každá SQL operácia musí byť vykonaná individuálne databázovým serverom. To znamená, že klientská aplikácia musí zaslať každý dotaz databázovému serveru samostatne, počkať kým ho spracuje, prijať a spracovať výsledky, urobiť výpočty a až potom odoslať ďalšie dotazy serveru. Toto všetko môže znamenať komunikáciu medzi viacerými procesmi a môže tiež znamenať zvýšenú sieťovú réžiu pokiaľ je klient na inom stroji ako databázový server.

S pgSQL je možné zoskupovať blok výpočtov a skupinu dotazov vo vnútri databázového serveru, čo znamená, že máme k dispozícií výpočtovú silu procedurálneho jazyka a zároveň jednoduchosť použitia SQL jazyka a navyše so značnou úsporou na nadbytočnej sieťovej réžií. pgSQL teda oproti SQL eliminuje nadbytočné oneskorenie medzi serverom a klientom, medzi-výsledky, ktoré klient nepotrebuje, nemusia byť prenášané medzi klientom a serverom a niekoľko-násobná syntaktická analýza dotazov môže byť vynechaná [16].

3.4.2 Partitioning v PostgreSQL

Ako sme už spomenuli, PostgreSQL je objektovo relačný databázový systém, čo používateľovi umožňuje použiť výhody dedenia. Treba poznamenať, že SQL:1999 štandard a neskoršie definujú určitú funkcionalitu dedenia, ktorá sa však v mnohých aspektoch líši od tej, ktorá je implementovaná v PostgreSQL. Pre správne pochopenie princípov dedenia a partitioningu si teraz definujeme niektoré základné pojmy.

Ako prvým je *rodičovská tabuľka*, ktorá je hlavnou tabuľkou z ktorej budú nové tabuľky dediť. Príkladom takejto tabuľky môže byť tabuľka miest mestá, ktorú definujeme nasledovne:

```
CREATE TABLE mesta (
    nazov        TEXT,
    obyvatelstvo FLOAT,
    vyska        INT
);
```

Detská tabuľka, nazývaná aj *partícia*, patrí pod rodičovskú tabuľku a zároveň dedí jej štruktúru. Nadväzujúc na rodičovskú tabuľku mestá si definujeme pomocou klauzuly INHERITS partíciu hlavne_mesta, ktorá bude dediť všetky stĺpce od rodičovskej tabuľky a bude mať jeden stĺpec navyše označujúci krajinu, ktorej hlavným mestom je.

```
CREATE TABLE hlavne_mesta (
    krajina CHAR(2)
) INHERITS (mesta);
```

Vďaka dedeniu je potom možné vykonávať dotazy nad rodičovskou tabuľkou čo spôsobí prehľadanie aj detských tabuliek. Nasledujúci dotaz by teda vrátil názvy miest a ich nadmorskú výšku a zahrnul by do výsledku aj mestá z tabuľky hlavných miest:

```
SELECT nazov, vyska
FROM mesta
WHERE vyska > 100;
```

Čo by vrátilo výsledok:

```
   nazov    |   vyska
-----------|----------
Bratislava |     152
Brno       |     190
Praha      |     177
Salzburg   |     424
Zlin       |     233
```

Pre vypísanie miest len z rodičovskej tabuľky by sme mohli použiť klauzulu ONLY nasledujúcim spôsobom:

```
SELECT nazov, vyska
FROM ONLY mesta
WHERE vyska > 100;
```

Čo by nám vrátilo len mestá z tabuľky mestá:

```
   nazov    |   vyska
-----------|----------
Brno       |     190
Salzburg   |     424
Zlin       |     233
```

Dedenie však nezabezpečuje automatické naplnenie dát z INSERT a COPY príkazov do iných tabuliek v hierarchií dedenia. Pri našom príklade by tento typ dotazu zlyhal:

```
INSERT INTO mesta (nazov, obyvatelstvo, vyska, krajina)
VALUES ('Vieden', NULL, NULL, 'AT');
```

Mohli by sme si myslieť, že dáta budú nejakým spôsobom presmerované do tabuľky hlavne_mesta, to sa však nestane. Príkaz INSERT vždy vkladá presne do tej tabuľky, do ktorej si to určíme. Existuje však možnosť ako dáta presmerovať do tabuľky hlavne_mesta a tou je trigger.

Trigger je sada uložených príkazov, ktoré sú asociované s konkrétnou tabuľkou a pri splnení určitej podmienky vykonajú špecifikovanú funkciu (v našom prípade partitioningu ju budeme volať rozdeľovacia funkcia). Trigger môže byť definovaný tak, aby reagoval pred tým ako sa daná operácia pokúsi vykonať zmenu na riadku (BEFORE), po operácií na riadku (AFTER), alebo namiesto danej operácie (INSTEAD OF) [17].

Rozdeľovacia (partition) funkcia je uložená procedúra, ktorá pri partitioningu určuje, do ktorej partície sa má uložiť nový záznam, resp. či je potrebné vytvoriť novú partíciu.

Pre náš prípad by sme mohli vytvoriť nasledujúci trigger:

```
CREATE TRIGGER mesta_trigger
BEFORE INSERT ON mesta
EXECUTE PROCEDURE rozdelovacia_funkcia;
```

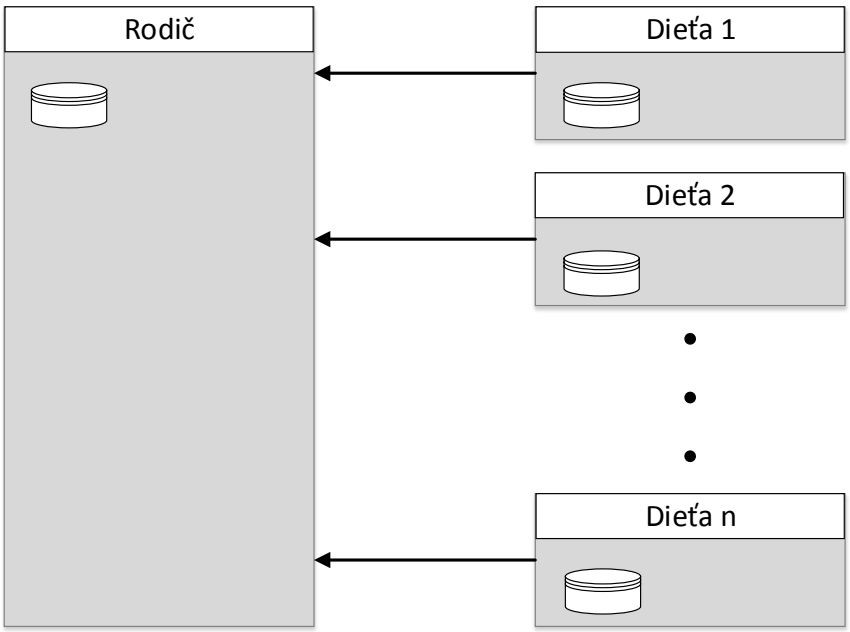
Následne by sme v rozdeľovacej funkcii určili, že pokiaľ je v dotaze INSERT uvedený aj stĺpec krajina, tento záznam sa vloží do detskej tabuľky hlavne_mesta.

Funkcionalita dedenia tiež zabezpečuje, že všetky CHECK obmedzenia (constraints) a NOT-NULL obmedzenia budú zdedené detskou tabuľkou od rodiča. Ostatné typy obmedzení (unikátny, primárny a cudzí kľúč) nie sú automaticky zdedené.

Spomenuté *CHECK obmedzenia* sú podmienky, ktoré musia byť splnené pre každý riadok tabuľky nad ktorou sú nastavené. PostgreSQL plánovač ich tiež používa na *constraint exclusion*, čo je optimalizačná technika využívaná pri partitioningu. Znamená to, že pokiaľ plánovač pri plánovaní vykonania dotazu nájde na danej tabuľke CHECK obmedzenie, ktoré nespĺňa podmienky dotazu, potom pri vyhľadávaní výsledkov túto tabuľku preskočí a urýchli tým vykonanie daného dotazu [18].

Po definovaní základných pojmov teraz popíšeme princíp partitioningu. Partitioning môžeme rozdeliť na dva typy:

- *Horizontálny partitioning* vkladá riadky rodičovskej tabuľky do partícií podľa určitých kritérií
- *Vertikálny partitioning* spočíva vo vytváraní partícií s menším počtom stĺpcov ako má rodičovská tabuľka. Menej používané stĺpce môžu byť vložené do nových partícií, ktoré sú následne uložené na pomalšie zariadenie



Obrázek 3.4: Princíp horizontálneho partitioningu

Pri horizontálnom partitioningu sa jedna veľká rodičovská tabuľka rozdelí na menšie celky (obrázok 3.4) - partície, ktoré od nej dedia jednotlivé stĺpce.

Dáta z veľkej rodičovskej tabuľky sa tým rozdistribuujú do jednotlivých menších tabuliek čo spôsobí, že práca s jednotlivými partíciami bude jednoduchšia ako práca nad veľkou rodičovskou tabuľkou.

Dáta sa pri partitioningu môžu rozdeliť dvomi spôsobmi:

- Podľa príslušnosti k rozsahu - môžu byť rozdelené napríklad podľa ID (partície rozsahov 1 - 100000, 100101 - 200000, atď), alebo napríklad podľa dátumu (partície rozdelené podľa dní 2014_04_01, 2014_04_02, atď)
- Podľa príslušnosti k zoznamu - napríklad príslušnosť k danému mestu (Brno, Praha, Bratislava) alebo ku kategóriám (Programovanie, Domácnosť, Jedlo)

V tejto kapitole sme spomenuli, že plánovač využíva CHECK obmedzenia na zrýchlenie vykonávania dotazov, v nasledujúcej kapitole si popíšeme ako plánovanie dotazov prebieha.

3.4.3 Plánovanie dotazov

PostgreSQL plánovač slúži na vytvorenie najoptimálnejšieho vykonania dotazu. Daný SQL dotaz môže byť vykonaný viacerými spôsobmi, z ktorých každý produkuje rovnaký výsledok. Plánovač preskúma každý z možných spôsobov vykonania a vyberie ten, o ktorom predpokladá, že sa vykoná najrýchlejšie. Postup akým plánovač spracúva dotaz je nasledovný: PostgreSQL server preberie z klientskej aplikácie dotaz. Následne je dotaz odovzdaný syntaktickému analyzátoru, ktorý skontroluje či sa v dotaze nenachádzajú syntaktické chyby. Ak je dotaz syntakticky správny, analyzátor prevedie dotaz na syntaktický strom. Analyzátor následne odovzdá syntaktický strom plánovaču. Plánovač traverzuje celým stromom a nájde všetky možné cesty vykonania dotazu, ku ktorým následne priradí odhadované ceny vykonania. Z týchto ciest je potom vybraná tá s najnižšou cenou. Cesta s najnižšou cenou je následne prebraná vykonávateľom, ktorý postupne vyberie riadky z tabuľky tým spôsobom ako je to definované vo vybranej ceste a vráti ich ako výsledok dotazu.

Pre lepšie pochopenie ako sa daný dotaz vykonáva a jeho prípadnú optimalizáciu existuje v PostgreSQL príkaz EXPLAIN. Najčastejšie sa EXPLAIN využíva na SELECT dotazoch, je možné ho však použiť aj na INSERT, UPDATE, DELETE, EXECUTE alebo DECLARE dotazy. Okrem volania samotného EXPLAIN, ktoré len analyzuje daný dotaz a zobrazí ako sa pravdepodobne bude dotaz vykonávať môžeme ešte dodať klauzulu ANALYZE. Tá daný dotaz aj vykoná (prejavia sa zmeny na reálnej databáze) a zobrazí výsledky aj s aktuálnym časom vykonávania [19]. Ako EXPLAIN funguje si ukážeme na nasledujúcom dotaze:

```
SELECT priezvisko
FROM zamestnanci
WHERE plat >= 50000;
```

Použitím EXPLAIN vidíme ako by PostgreSQL daný dotaz vykonával:

```
EXPLAIN SELECT priezvisko
FROM zamestnanci
WHERE plat >= 50000;
                        QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on zamestnanci (cost=0.00..16.50 rows=173 width=118)
   Filter: (plat >= 50000)
```

Po použití EXPLAIN ANALYZE na tabuľke s 2 miliónmi riadkov by bol výsledok nasledovný:

```
EXPLAIN ANALYZE SELECT priezvisko
FROM zamestnanci
WHERE plat >= 50000;
                        QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on zamestnanci (cost=0.00..35811.00 rows=1 width=6) (actual
 time=2.401..295.247 rows=1428 loops=1)
   Filter: (plat >= 50000)
Total runtime: 295.379 ms
(3 rows)
```

Seq Scan on zamestnanci znamená, že na prehľadanie tabuľky bolo použité sekvenčné prehľadávanie. Prvá hodnota za cost určuje odhadovanú cenu, za ktorú sa podarí získať prvý riadok. Druhá hodnota určuje cenu, za ktorú sa podarí získať všetky riadky. Tieto hodnoty sú uvedené v jednotke rovnej sekvenčnému prehľadaniu jednej stránky (disk page fetch). Pre sekvenčné prehľadanie je táto hodnota 1.0. Hodnota rows je počet riadkov, ktoré plánovač očakáva, že budú vrátené pri použití daného prehľadávania. Width určuje priemernú šírku jedného riadku v bajtoch z výsledných dát. Pri actual time nám prvé dve hodnoty vyjadrujú podobne ako pri cost cenu kedy sa podarí získať prvý a všetky riadky, tento krát je to však reálna hodnota vykonania v milisekundách. Rows je pri actual time je taktiež reálny počet získaných riadkov vrátených pri danom prehľadávaní. Loops hodnota určuje koľko krát bolo dané prehľadávanie iterované.

V tejto kapitole sme si vysvetlili niektoré vlastnosti PostgreSQL, ktoré budú potrebné v ďalších častiach práce. V nasledujúcej kapitole sa budeme venovať konkrétnym tabuľkám Zabbix serveru v PostgreSQL databázy

3.5 Tabuľky databázy

Jednotlivé tabuľky v databáze systému Zabbix môžeme rozdeliť na 2 typy:

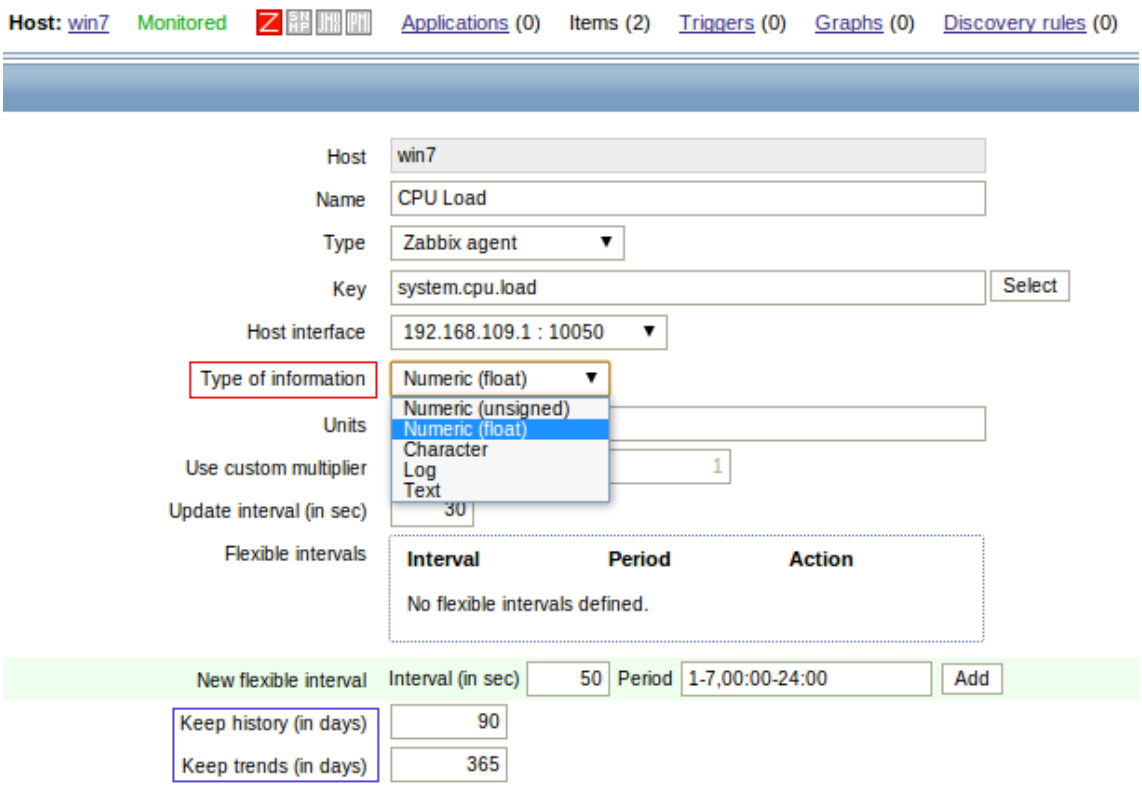
- dátové
- konfiguračné

Zameriame sa len na dátové tabuľky, keďže tie, ako ukážeme ďalej, obsahujú veľké množstvo záznamov a sú teda vhodným kandidátom pre optimalizáciu. Tabuľky s najväčším počtom záznamov zvyčajne sú: history_*, trends_* a events [13].

3.5.1 Tabuľky history_*

Tabuľky history slúžia na ukladanie hodnôt nameraných metrík. Zabbix databáza používa niekoľko tabuliek history označených suffixom _* kde * predstavuje typ ukladaných dát. Typ jednotlivých dát je možné nastaviť vo webovom rozhraní pri vytváraní novej sledovanej metriky po kliknutí na položky Configuration - Hosts - New Item - Type of Information ako vidíme na obrázku 3.5. Ak napríklad nastavíme typ metriky na Numeric (float), monitorované hodnoty tejto metriky sa budú ukladať do tabuľky history. Typy jednotlivých history tabuliek sú:

- history - numerické dáta (float)
- history_uint - numerické dáta (unsigned integer)
- history_str - znaky (do 255 bajtov)
- history_log - logy
- history_text - textové dáta



Obrázek 3.5: Nastavenie typu metriky

Nový záznam je pre konkrétnu metriku pridávaný podľa nastavenia Update interval. History tabuľky zvyčajne obsahujú najviac záznamov, keďže sa nová hodnota pridáva každých niekoľko sekúnd, čo môže pri veľkej topológii a veľa metrikách znamenať pridávanie niekoľko sto záznamov za sekundu. Na obrázku 3.6 môžeme vidieť štruktúru tabuľky history. Štruktúry medzi jednotlivými history tabuľkami sa okrem typu metriky nemenia.

	itemid [bigint]	**clock** [integer]	**value** [numeric(16,4)]	**ns** [integer]
1	23328	1357319778	0.2000	951998806
2	23328	1357319808	0.3500	46935655
3	23328	1357319838	0.2833	70828428

Obrázek 3.6: Štruktúra tabuľky history s Update interval = 30 sekúnd pre itemid = 23328

Význam jednotlivých stĺpcov v history tabuľke je nasledovný:

- itemid - unikátne číslo označujúcu konkrétnu metriku na konkrétnom agentovi
- clock - čas získania záznamu od agenta vo formáte Unix timestamp
- value - hodnota meranej metriky
- ns - počet nanosekúnd pri získaní záznamu (ide o spresnenie záznamov, keďže Unix timestamp je uvedený len v sekundách)

3.5.2 Tabuľky trends_* a events

Zatiaľčo tabuľka history uchováva každú nameranú hodnotu, pri tabuľke trends Zabbix server každú hodinu vypočíta z history tabuľky minimálne, maximálne a priemerné hodnoty metrík. Tabuľka trends teda obsahuje menej záznamov ako history. Do tabuľky trends sa ukladajú len číselné hodnoty, pre znaky, reťazce a logy sa tabuľky trends nepoužívajú.

Tabuľka events má uložené všetky udalosti, ktoré boli vygenerované podľa postupu popísaného v kapitole 3.2 Tok informácií.

3.6 Veľkosť databázy

V prípade rozsiahlej topológie a veľkého počtu sledovaných metrík môžu tabuľky history, trends a events dosiahnuť až stovky Gigabajtov a niekoľko miliónov záznamov. Veľkosť tabuľky záleží od nasledujúcich premenných:

Počet pridaných záznamov za sekundu
V prípade, že monitorujeme 3000 metrík a každú metriku aktualizujeme raz za 60 sekúnd, počet spracovaných hodnôt za sekundu je 3000/60 = 50. Znamená to, že každú sekundu je pridaných do databázy 50 záznamov.

Nastavenia pre dobu uchovania history záznamov
Ak by sme chceli uchovávať záznamy v tabuľke history po dobu 30 dní a počet spracovaných hodnôt za sekundu je 50, celkový počet hodnôt bude (30*24*3600)*50 = 129,6 miliónov hodnôt. V závislosti od typu hodnôt (float, integer, string, log súbory, atď.) a použitého

databázového systému sa veľkosť potrebná pre uloženie jedného záznamu môže líšiť od 40 až po stovky bajtov. Najčastejšie je to 50 bajtov. V našom prípade by bolo potrebné pre uloženie 129,6 milióna údajov 129,6 * 50 = 6,48 Gigabajtov, čo nie je veľa, avšak najväčším problémom je práve spomínaných 130 miliónov záznamov, čo robí prácu nad takou veľkou tabuľkou náročné na čas a výkon.

Nastavenia pre dobu uchovania trends záznamov
Trends záznamy sa pridávajú každú hodinu a tento parameter nie je možné zmeniť. Veľkosť pre jeden trend záznam je približne 128 bajtov, takže ak by sme chceli uchovávať údaje v trend tabuľke po dobu 5 rokov, pre 3000 metrík potrebujeme (3000/3600)*(24*3600*365)*128 = 3.4 Gigabajtov priestoru za rok a teda 16.8 Gigabajtov za 5 rokov.

Nastavenia pre dobu uchovania events záznamy
Každá udalosť potrebuje približne 130 bajtov miesta na disku. Je ťažké odhadnúť počet udalostí, ktoré budú denne v systéme Zabbix vygenerované, keďže to priamo súvisí s konkrétnymi sledovanými zariadeniami a ich správaním. V prípade, že by Zabbix generoval 1 udalosť za sekundu a chceli by sme uložiť záznamy o udalostiach po dobu 3 rokov, potrebovali by sme na to 3*365*24*3600 = 12.3 Gigabajtov miesta na disku [13].

Celková veľkosť databázy môže byť teda vypočítaná spočítaním konfiguračných tabuliek a tabuliek history, trends a events. Táto veľkosť nie je dosiahnutá ihneď po inštalácií nástroja Zabbix. Zväčšuje sa s časom a rýchlosť zväčšovania záleží najmä na počte sledovaných metrík a na nastavení funkcie Housekeeper, ktorú popíšeme v nasledujúcej kapitole.

3.7 Housekeeper

Ako sme uviedli v predchádzajúcej kapitole, Zabbix databáza môže dosahovať veľkosť až niekoľko desiatok Gigabajtov. Zabbix preto obsahuje funkciu na automatické zmazanie záznamov nazvanú *housekeeper*. Záznamy history a trends sú uložené dovtedy, kým ich housekeeper nezmaže. Ako vidíme na obrázku 3.5, pre každú metriku je možné meniť ich dobu uchovania. Pri základných nastaveniach je housekeeper spúšťaný raz za hodinu, čo bolo možné do Zabbix verzie 2 zmeniť v súbore zabbix_server.conf zmenou hodnoty parametra HousekeepingFrequency=1, ktorá je uvedená v hodinách. Funkciu housekeeper je tiež možné vypnúť a to nastavením hodnoty parametra DisableHousekeeping na 1. Od verzie 2 sa housekeeper nastavuje vo webovom rozhraní. Vypnutie housekeepera však znamená neustále pribúdanie záznamov v tabuľkách a teda eventuálne zaplnenie disku.

Automatické mazanie záznamov v databáze systému Zabbix sa teda môže po popise housekeepera zdať ako bezproblémové, housekeeper má však veľkú nevýhodu a tou je spôsob mazania jednotlivých záznamov. V zdrojovom kóde súboru housekeeper.c môžeme vidieť ako funkcia delete_history, ktorá je určená na mazanie tabuliek history a trends pracuje:

```
static int delete_history(const char *table, zbx_uint64_t itemid, int
keep_history, int now)
{
   const char *__function_name = "delete_history";
   DB_RESULT result;
   DB_ROW row;
```

```
    int min_clock, deleted;

    zabbix_log(LOG_LEVEL_DEBUG, "In %s() table:'%s' itemid:" ZBX_FS_UI64 "
    keep_history:%d now:%d",__function_name, table, itemid, keep_history, now);

    result = DBselect("select min(clock) from %s where itemid=" ZBX_FS_UI64,
    table, itemid);

    if (NULL == (row = DBfetch(result)) || SUCCEED == DBis_null(row[0])) {
      DBfree_result(result); return 0; }
    min_clock = atoi(row[0]);
    min_clock = MIN(now - keep_history * SEC_PER_DAY, min_clock + 4 *
    CONFIG_HOUSEKEEPING_FREQUENCY * SEC_PER_HOUR);
    DBfree_result(result);

    deleted = DBexecute("delete from %s where itemid="ZBX_FS_UI64 "and clock<%d",
    table, itemid, min_clock);
    zabbix_log(LOG_LEVEL_DEBUG, "End of %s():%d", __function_name, deleted);

    return deleted;
}
```

V zvýraznenom kóde vidíme, že funkcia využíva na zmazanie záznamov SQL operáciu DELETE a to na každý jeden záznam. V prípade, že máme nastavené 30-dňové uchovávanie hodnôt a je pridaných 50 záznamov za sekundu, na zmazanie týchto záznamov housekeeper musí vykonať približne 130 miliónov operácií DELETE [13].

3.7.1 Nevýhody funkcie housekeeper

Spôsob akým funkcia housekeeper odstraňuje nepotrebné záznamy predstavuje pre databázový server dva problémy:

- vysoká záťaž CPU
- neuvoľnenie priestoru na disku po operácií DELETE

V prvom prípade môže mazanie miliónov riadkov zaťažiť CPU až na 100%, čo môže viesť k situácií kedy databázový server nebude schopný zapisovať dáta nových metrík. Môže teda nastať situácia, že monitorované zariadenie nie je funkčné, Zabbix to pre vysoké zaťaženie databázy nedetekuje a správca nebude o tejto situácií vedieť.

Druhým problémom je neuvoľnenie priestoru po zmazaných záznamoch operáciou DELETE. PostgreSQL ako aj iné databázové systémy pri UPDATE alebo DELETE operácií ponechávajú starú verziu daného riadku uloženú na disku, aby bolo možné využiť výhody súbežného riadenia (MVCC). Stará verzia riadku preto nesmie byť odstránená kým je ešte

potencionálne prístupná z iných transakcií. Ak by databáza nezabezpečovala súbežné riadenie, mohlo by dôjsť k prípadu, kde by jedna transakcia prepísala dáta čítané inou transakciou.

Neuvoľnenie priestoru po zmazaných záznamoch začína byť problematické v prípade keď je pridávaný a odoberaný veľký počet záznamov. Veľkosť databázy sa vtedy postupne zväčšuje až do zaplnenia disku, čo opäť spôsobí znefunkčnenie databázového systému. PostgreSQL preto obsahuje operáciu VACUUM.

VACUUM odstraňuje staré verzie riadkov a znovu umožní na toto miesto zapisovať nové dáta. Nevráti však miesto na disku späť operačnému systému. Na to je potrebná operácia VACUUM FULL, ktorá vytvorí novú verziu súboru danej tabuľky, bez zbytočného voľného, nepoužitého miesta.

Používanie VACUUM FULL rieši problém zväčšujúcej sa tabuľky, vyžaduje však exkluzívny prístup k celej tabuľke, takže počas jej behu nie je možné pridávať nové hodnoty metrík z meraných zariadení. Okrem toho tiež významne zaťažuje CPU a ešte viac zhoršuje prvý problém pri vykonávaní DELETE operácií [20].

Zo spomenutých dôvodov je funkcia housekeeper vhodná len pre siete s malým počtom zariadení a monitorovaných metrík. Pre väčšie topológie je preto potrebné navrhnúť riešenie, ktoré bude zahŕňať oba problémy spomenuté v tejto kapitole.

Kapitola 4

Stav Zabbix serveru pred optimalizáciou

V predchádzajúcich kapitolách sme podrobne popísali funkcionalitu monitorovacieho nástroja Zabbix a uviedli sme niektoré dôležité princípy z PostgreSQL databázy. V tejto kapitole popíšeme VUT sieť s ktorou Zabbix pracuje a pomocou dát získaných priamo Zabbix serverom vysvetlíme v čom spočíva najväčší problém pri používaní tohto nástroja v rozsiahlych topológiách. Informácie o činnosti Zabbixu v grafoch uvádzame pred optimalizáciou systému, aby bolo možné ich po optimalizácií porovnať a následne tak ukázať či bola optimalizácia efektívna, alebo nie.

Pre predstavu v akom prostredí je systém Zabbix nasadený si v tejto časti predstavíme sieť VUT, ktorú spravujú pracovníci Centra výpočetních a informačních služeb VUT (CVIS). Ide o 10Gbitovú chrbticovú sieť VUT v Brne, ktorá je pripojená do akademickej siete CESNET. VUT sieť zahŕňa prvky ako sieťové prepínače, záložné zdroje, rôzne druhy serverov, sieťové tlačiarne a ďalšie iné sieťové prvky. Od roku 2005 CVIS taktiež realizuje správu študentskej siete KolejNet, na ktorú je pripojených viac než 7000 klientských staníc.

Z pohľadu vyťaženia Zabbix serveru si v nasledujúcej tabuľke uvedieme niektoré parametre VUT siete.

Tabulka 4.1: Kľúčové parametre Zabbix serveru v sieti VUT

Názov	**Hodnota**
Počet monitorovaných zariadení	97
Počet sledovaných metrík	3221
Počet triggerov	1688
Počet pridaných záznamov za sekundu	40.26

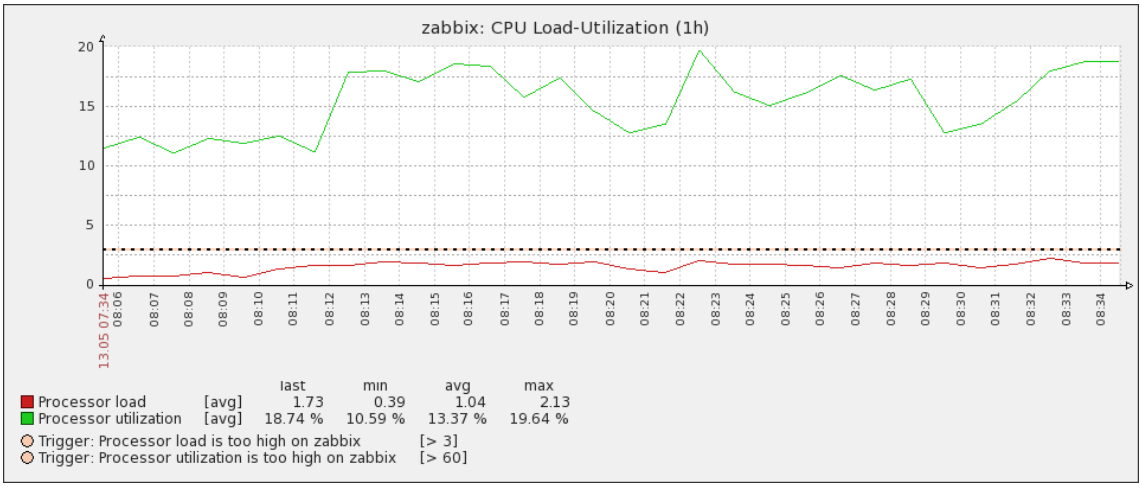
Z tabuľky 4.1 vidíme hlavný indikátor zaťaženia Zabbix serveru, ktorým je počet pridaných záznamov za sekundu, ktorý sme popísali v kapitole 3.6. Dáta zo zariadení sú získavané buď použitím Zabbix agenta, alebo pomocou protokolu SNMP, resp. z oboch súčasne.

Veľkosť VUT siete sa prejaví v nasledujúcej tabuľke, ktorá zobrazuje akú veľkosť súčasná databáza Zabbix serveru pred optimalizáciou v sieti VUT dosahuje. Údaje v tabuľkách history_* sú uchovávané po dobu 365 dní a ostatné tabuľky sú uchovávané od počiatku bez premazávania.

Tabulka 4.2: Údaje o veľkosti databázy Zabbix serveru v sieti VUT

Názov	**Hodnota**
Celková veľkosť PostgreSQL databázy Zabbix serveru	119 GB
Počet riadkov najväčšej tabuľky history	537,806,592
Počet tabuliek prekračujúcich 5 miliónov riadkov	7

V tabuľke 4.2 vidíme, že najväčšia tabuľka history dosahuje viac ako 500 miliónov riadkov, čo pri práci s touto tabuľkou spôsobuje značné problémy. Napríklad pri pokuse o náhľad riadkov takejto tabuľky stúpne zaťaženie CPU o 10% (čas 8:12 na obrázku 4.1), celé webové rozhranie phpPgAdmin zamrzne na niekoľko minút a k načítaniu riadkov vôbec nedôjde.



Obrázek 4.1: Zvýšené záťaže CPU pri práci s najväčšou history tabuľkou

Keďže bol Zabbix server v sieti VUT v činnosti už niekoľko mesiacov, za tú dobu sa v ňom nahromadilo veľké množstvo dát. V ďalšej časti uvedieme východzí stav Zabbix serveru a jeho nastavení, z ktorého sme vychádzali.

Keďže je funkcia Housekeeper vo veľkých sieťach akou je VUT prakticky nevyužiteľná, pracovníci CVIS si vytvorili vlastný perl skript, ktorým mazali staré, nepotrebné záznamy z databázy.

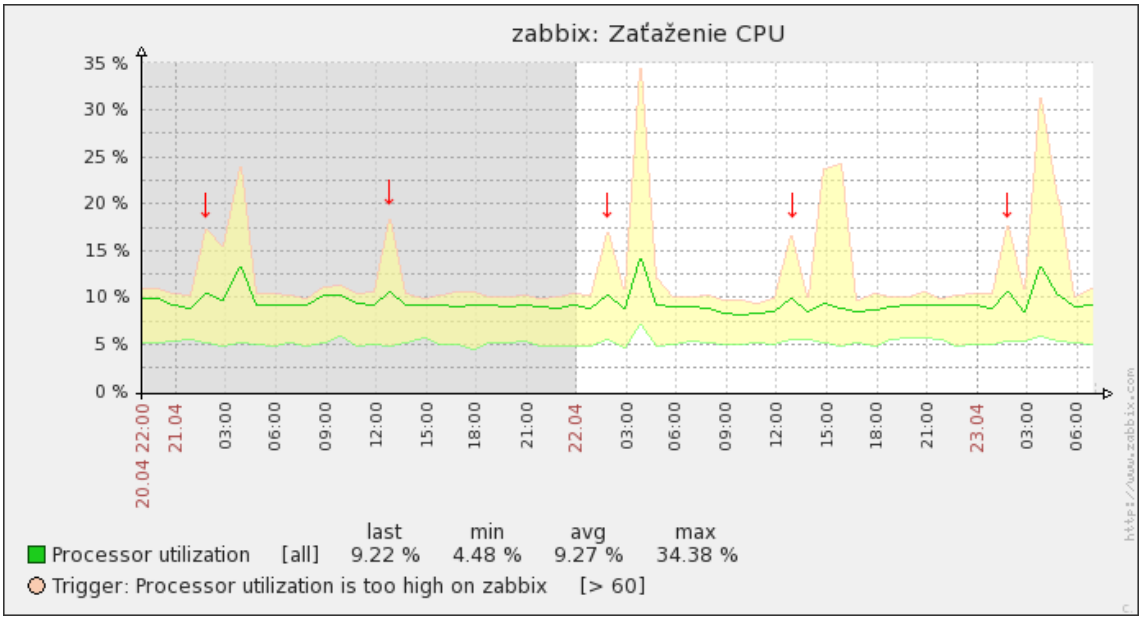
Tento skript sa spúšťal dva-krát denne pomocou cronu a bol nastavený na zmazanie záznamov starších ako 365 dní z tabuliek history_*. Z vybranej časti kódu tohto skriptu môžeme vidieť princíp mazania jednotlivých záznamov z tabuliek history_*.

```
my $NDAYS = 365;
my $ts = time() - $NDAYS * 60 * 60 * 24;
query("DELETE FROM history WHERE clock < $ts");
query("DELETE FROM history_uint WHERE clock < $ts");
query("DELETE FROM history_str WHERE clock < $ts");
query("DELETE FROM history_log WHERE clock < $ts");
```

```
query("DELETE FROM history_text WHERE clock < $ts");
```

Skript podobne ako Housekeeper vykonával SQL operáciu DELETE nad tými riadkami databázy, ktoré spĺňali podmienku, že sú staršie ako 365 dní. Opäť teda dochádzalo k zvýšeniu zaťaženia CPU a neuvoľňovaní miesta na disku. Na obrázku 4.2 môžeme vidieť šípkami označené, periodicky sa opakujúce, zvýšenia zaťaženia CPU v čase 2:39 a 13:39 spôsobené spomínanou operáciou DELETE.

Pre objasnenie spomenieme čo znamená zvýšenie zaťaženia v čase po 3:00. Toto zaťaženie je spôsobené skriptom, ktorý kontroluje integritu súborového systému a vytvára zálohy. Cieľom tejto práce však nebude optimalizácia tohto skriptu.



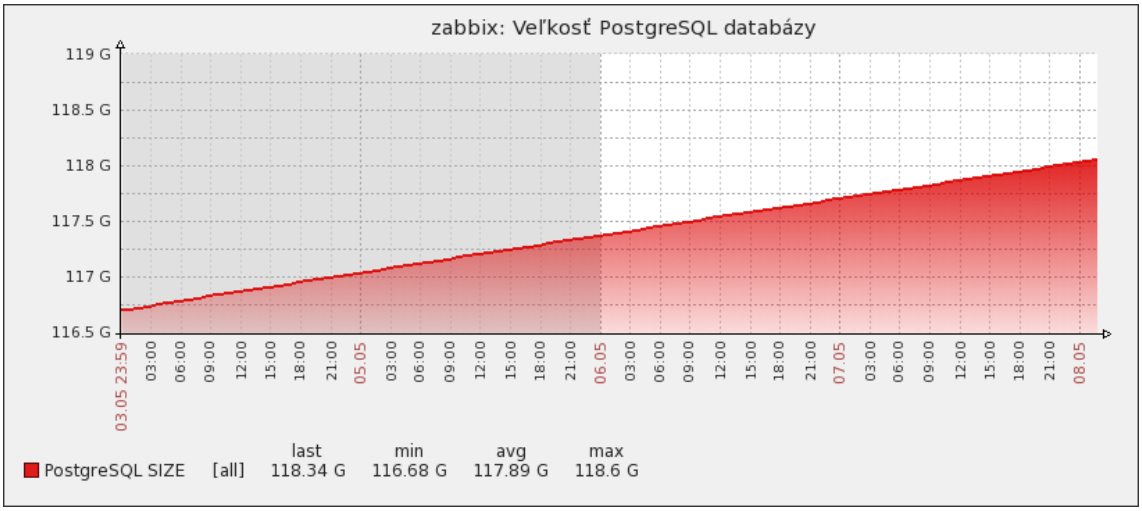
Obrázek 4.2: Vplyv skriptu využívajúceho operáciu DELETE na CPU

Pre získanie prehľadu o aktuálnom stave PostgreSQL databázy pred optimalizáciou sme použili parametre tup_deleted, tup_updated, tup_inserted, pg_database_size, ktoré poskytuje subsystém PostsgreSQL zodpovedný za zber štatistík [21]. Tieto parametre je možné sledovať v prostredí Zabbix po pridaní nasledovných riadkov definujúcich nové používateľské parametre na koniec konfiguračného súboru zabbix agenta:

```
UserParameter=psql.db_deleted[*],psql -t -c "select tup_deleted from
pg_stat_database where datname = '$1'"
UserParameter=psql.db_updated[*],psql -t -c "select tup_updated from
pg_stat_database where datname = '$1'"
UserParameter=psql.db_inserted[*],psql -t -c "select tup_inserted from
pg_stat_database where datname = '$1'"
UserParameter=psql.db_size[*],psql -t -c "select pg_database_size('$1')"
```

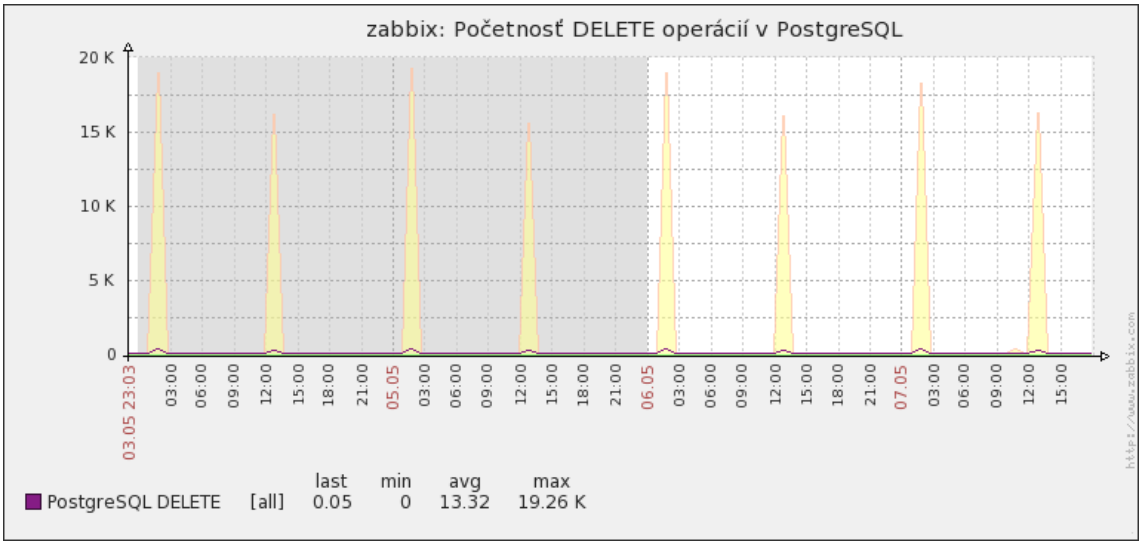
Následne sa pridá nová metrika pre daný uzol s názvom nového používateľského parametra (napr. psql.db_size[*]) a metriku je potom možné zobrazovať v ľubovolných Zabbix grafoch. Z grafu vytvoreného z týchto údajov môžeme na obrázku č. 4.3 vidieť, že sa veľ-

kosť PostgreSQL databázy na disku po odstránení riadkov operáciou DELETE skutočne nezmenšuje.



Obrázek 4.3: Veľkosť PostgreSQL databázy pred optimalizáciou

Pred optimalizáciou bolo taktiež zaradené sledovanie počtu DELETE operácií, ktoré sa vykonávajú pri neefektívnom mazaní nepotrebných riadkov z tabuliek history_*. Ako vidno na obrázku 4.4, pri jednom spustení skriptu sa vykoná takmer 20 000 operácií DELETE.



Obrázek 4.4: Veľkosť PostgreSQL databázy pred optimalizáciou

Tabuľky trends_* sa pre uchovanie dlhodobých záznamov o stave siete nezmazávajú a bolo ich potrebné ponechať aj v budúcnosti, po optimalizácií systému.

V kapitole 3.4.3 sme opísali ako je možné pomocou EXPLAIN pochopiť plánovanie jednotlivých dotazov. Tieto poznatky teraz použijeme pre získanie prehľadu o tom, ako plánovač postupuje pri prehľadávaní tabuliek pred optimalizáciou. Pozrime sa ako plánovač postupuje pri nasledujúcom dotaze nad history tabuľkou:

```
EXPLAIN ANALYZE SELECT *
FROM history
WHERE clock > 1394841600 AND clock < 1395014400
```

Z tabuľky history vyberáme všetky riadky, ktoré splňujú podmienku, že boli vložené v dátume od 15.03.2014 do 17.03.2014. Výsledok EXPLAIN nie je prekvapujúci:

```
                        QUERY PLAN
-----------------------------------------------------------------
Seq Scan on history (cost=0.00..926510.60 rows=2839499 width=21) (actual
time=636.116..5824.925 rows=2858660 loops=1)
  Filter: ((clock > 1394841600) AND (clock < 1395014400))
  Rows Removed by Filter: 40412381
Total runtime: 5928.246 ms
```

Vidíme, že na prehľadanie a odfiltrovanie riadkov pridaných v určenom čase plánovač používa sekvenčné prehľadanie na celej tabuľke history, čo trvá takmer 6 sekúnd.

Kapitola 5

Optimalizácia schématu databázy

Hlavnou myšlienkou optimalizácie PostgreSQL databázy je nahradiť funkciu Housekeeper a jej náročné mazanie starých záznamov využívajúce operáciu DELETE. Optimalizácia databázy sa teda musí vyhýbať použitiu tejto SQL operácie.

Druhou možnosťou ako zmazať záznamy v PostgreSQL databáze je zmazať celú tabuľku. Na zmazanie tabuľky je možné použiť SQL funkciu DROP. Ak by sme však zmazali napríklad celú tabuľku history, neplnilo by to svoj účel, keďže by sme prišli aj o aktuálne údaje, ktoré by sme ešte v budúcnosti mohli potrebovať. Práve mazanie automaticky vytvorených partícií pomocou operácie DROP využívajú obe riešenia, ktoré v tejto kapitole uvedieme.

Keďže Zabbix neponúka žiadne oficiálne riešenie automatického partitioningu, komunita okolo systému Zabbix si vytvorila svoje riešenia, ktoré si teraz popíšeme. V súčasnosti existujú dve riešenia, jedno z nich vyžaduje prázdnu databázu a druhé je možné implementovať na zabehnutý Zabbix server s už existujúcimi údajmi. Každé z nich má svoje výhody a nevýhody, preto si zvolíme to najvhodnejšie riešenie pre VUT sieť. Pre názornosť budeme zobrazovať len hlavné časti kódu, ktorých význam si popíšeme.

5.1 Riešenie vyžadujúce prázdnu databázu

Toto riešenie [22] má jeden veľký nedostatok a tým je potrebná prázdna Zabbix databáza pri prvom spustení skriptu. Pri tomto riešení je potrebné databázu najskôr inicializovať spustením funkcie create_zbx_partitions.

5.1.1 Vytváranie partícií

Funkcia create_zbx_partitions má za úlohu vytváranie nových partiícií. Funkcia si najskôr vytvorí pomocné tpl_ tabuľky, ktoré sú potrebné pre jednoduchšie generovanie partícií s potrebnými indexami. Následne odstráni všetky indexy a obmedzenia z pôvodných tabuliek:

```
IF init > 0 THEN
   EXECUTE 'create table if not exists tpl_' || part || '(like ' || part || '
   including defaults including storage including constraints including indexes)';
   FOR cons IN SELECT constraint_name FROM information_schema.table_constraints
   WHERE table_name = part AND constraint_type = 'PRIMARY KEY' LOOP
```

```
        EXECUTE 'ALTER TABLE ' || part || ' DROP CONSTRAINT IF EXISTS ' ||
        quote_ident(cons.constraint_name) || ' CASCADE';
    END LOOP;
    FOR name IN SELECT * FROM pg_indexes WHERE tablename = part LOOP
        EXECUTE 'DROP INDEX ' || quote_ident(name.indexname);
    END LOOP;
 END IF;
```

Funkcia ďalej určí, či bola zavolaná pre tabuľku, na ktorej sa bude partitioning vykonávať denne, alebo mesačne.

```
IF part_mode = 'daily' THEN
   IF init > 0 THEN
      SELECT EXTRACT(epoch FROM date_trunc('day',CURRENT_TIMESTAMP)) INTO
      current_check;
      SELECT EXTRACT(epoch FROM date_trunc('day',CURRENT_TIMESTAMP + INTERVAL '1
      day')) INTO next_check;
      SELECT TO_CHAR(CURRENT_TIMESTAMP ,'_yyyymmdd') INTO next_partition;
   ELSE
      SELECT EXTRACT(epoch FROM date_trunc('day',CURRENT_TIMESTAMP + INTERVAL '1
      day')) INTO current_check;
      SELECT EXTRACT(epoch FROM date_trunc('day',CURRENT_TIMESTAMP + INTERVAL '2
      day')) INTO next_check;
      SELECT TO_CHAR(CURRENT_TIMESTAMP + INTERVAL '1 day','_yyyymmdd') INTO
      next_partition;
   END IF;
END IF;
IF part_mode = 'monthly' THEN
   IF init > 0 THEN
      SELECT EXTRACT(epoch FROM date_trunc('month',CURRENT_TIMESTAMP)) INTO
      current_check;
      SELECT EXTRACT(epoch FROM date_trunc('month',CURRENT_TIMESTAMP + INTERVAL
      '1 month')) INTO next_check;
      SELECT TO_CHAR(CURRENT_TIMESTAMP ,'_yyyymm') INTO next_partition;
   ELSE
      SELECT EXTRACT(epoch FROM date_trunc('month',CURRENT_TIMESTAMP + INTERVAL
      '1 month')) INTO current_check;
      SELECT EXTRACT(epoch FROM date_trunc('month',CURRENT_TIMESTAMP + INTERVAL
      '2 month')) INTO next_check;
```

```
    SELECT TO_CHAR(CURRENT_TIMESTAMP + INTERVAL '1 month','_yyyymm') INTO
    next_partition;
  END IF;
END IF;
```

Podľa toho funkcia skontroluje, či už tabuľka pre daný deň/mesiac existuje, alebo ju treba vytvoriť. Funkcia taktiež vytvára pohotovostnú (emergency) partíciu, pre každú jednu rodičovskú tabuľku. Do nej sa vloží záznam vtedy, keď potrebná partiícia pre daný deň/mesiac ešte neexistuje.

```
created_partition:='partitions.' || part || next_partition;
EXECUTE 'create table if not exists ' || created_partition ||' (check ( clock >=
' || current_check || ' and clock < ' || next_check || '),like tpl_' || part
 || ' including defaults including storage including constraints including
 indexes) inherits(' || part || ')';
EXECUTE 'create table if not exists partitions.emergency_' || part || '( like '
|| part || ' including defaults including storage including constraints including
indexes) inherits(' || part || ')';
```

5.1.2 Vytvorenie triggeru pre každú tabuľku

Samotné vkladanie záznamov je zabezpečené pomocou triggeru, nastaveného na jednotlivé rodičovské tabuľky:

```
CREATE TRIGGER history_trigger
BEFORE INSERT ON history
FOR EACH ROW
EXECUTE PROCEDURE dynamic_insert_trigger('daily');
```

Pred vložením záznamu do rodičovskej tabuľky, sa vždy vykoná funkcia dynamic_insert_trigger().

5.1.3 Vkladanie záznamov

Funkcia dynamic_insert_trigger() podľa argumentu zistí či ide o dennú, alebo mesačnú tabuľku a následne vloží záznam do potrebnej partície:

```
IF TG_ARGV[0] = 'daily' THEN
   timeformat:='_yyyymmdd';
ELSIF TG_ARGV[0] = 'monthly' THEN
   timeformat:='_yyyymm';
END IF;
EXECUTE 'INSERT INTO partitions.' || TG_TABLE_NAME ||
TO_CHAR(TO_TIMESTAMP(NEW.clock),timeformat) || ' SELECT ($1).*' USING NEW;
RETURN NULL;
```

```
EXCEPTION
   WHEN undefined_table THEN
   EXECUTE 'INSERT INTO partitions.emergency_' || TG_TABLE_NAME || ' SELECT
   ($1).*' USING NEW;
   RETURN NULL;
END;
```

Ak potrebná partícia partiícia ešte neexistuje, záznam sa vloží do pohotovostnej emergency partície, aby nedošlo k strate údajov.

Pre automatizáciu tohoto riešenia je použitý cron, ktorý denne volá funkciu create_zbx_partitions a vytvorí pre všetky definované tabuľky novu partíciu pre nasledujúci deň. Toto riešenie neobsahuje žiadnu funkciu, ktorá by staré partície po určitom čase zmazala. Je preto potrebné ich mazať manuálne, alebo si vytvoriť vlastnú funkciu na ich mazanie.

5.1.4 Analýza riešenia

Po popise tejto metódy môžeme konštatovať nasledujúce fakty:

- metóda potrebuje pre fungovanie prázdnu databázu, nie je možné ju použiť na zabehnutú Zabbix inštaláciu
- partície sa denne v určenom čase vytvárajú deň dopredu prostredníctvom cronu
- ak partícia do ktorej sa má záznam vložiť ešte nebola vytvorená, uloží sa záznam do pohotovostnej tabuľky, aby sa dáta nestratili
- neobsahuje funkciu mazania starých partícií

5.2 Riešenie nevyžadujúce prázdnu databázu

Nasledujúce riešenie [23] je možné aplikovať priamo na zabehnutú inštaláciu Zabbixu a teda nemá problém s databázou, ktorá už obsahuje záznamy. Jej funkcionalitu môžeme rozdeliť na niekoľko hlavných krokov.

1. Vytvorenie rozdeľovacej funkcie
2. Vytvorenie triggeru pre každú tabuľku
3. Vytvorenie funkcie, ktorá bude mazať staré partície
4. Nastavenie cronu, ktorý bude vykonávať funkciu z bodu 3
5. Migrácia starých záznamov do nových partícií

5.2.1 Rozdeľovacia funkcia

Rozdeľovacia funkcia trg_partition() slúži na vytvorenie nových partícií rozdelených podľa dní, alebo mesiacov. V tejto funkcii si najskôr z argumentu funkcie určíme, či budú vytvárané partície denne, alebo mesačne:

```
IF selector = 'day' THEN
   timeformat := 'YYYY_MM_DD';
ELSIF selector = 'month' THEN
   timeformat := 'YYYY_MM';
END IF;
```

Následne sa funkcia pokúsi vložiť nový záznam do partície pre súčasný deň, resp. mesiac. Využíva sa tu BEGIN EXCEPTION konštrukcia jazyka pgSQL, ktorá je podobná try - catch konštrukcií používanej v niektorých objektovo-orientovaných jazykoch.

```
EXECUTE 'INSERT INTO ' || prefix || quote_ident(tablename) || ' SELECT ($1).*'
USING NEW;
RETURN NULL;

EXCEPTION
WHEN undefined_table THEN
```

EXCEPTION konštrukcia zabezpečuje, že v prípade keď partícia ešte neexistuje, partíciu vytvoríme a vytvoríme pre ňu tiež kompozitný index zložený zo stĺpcov itemid a clock. Tabuľka events nepoužíva stĺpec itemid, použije sa preto v indexe eventid.

```
startdate := EXTRACT(epoch FROM date_trunc(selector, TO_TIMESTAMP(NEW.clock)));
enddate := EXTRACT(epoch FROM date_trunc(selector, TO_TIMESTAMP(NEW.clock) +
_interval ));

create_table_part:= 'CREATE TABLE IF NOT EXISTS '|| prefix ||
quote_ident(tablename) || ' (CHECK ((clock >= ' || quote_literal(startdate) || '
AND clock < ' || quote_literal(enddate) || ''))) INHERITS ('|| TG_TABLE_NAME ||
')';
create_index_part:= 'CREATE INDEX '|| quote_ident(tablename) || '_1 on ' ||
prefix || quote_ident(tablename) || '(itemid,clock)';

EXECUTE create_table_part;
EXECUTE create_index_part;

EXECUTE 'INSERT INTO ' || prefix || quote_ident(tablename) || ' SELECT ($1).*'
USING NEW;
RETURN NULL;
```

Dedenie novovytvorenej partície od hlavnej tabuľky je zabezpečené klauzulou INHERITS pri vytváraní partície. Po vytvorení partície sa záznam opäť pokúsi vložiť.

5.2.2 Vytvorenie triggeru pre každú tabuľku

Na vytváranie partícií rozdeľovacou funkciou je potrebné nastaviť na všetky tabuľky triggery, ktoré pred vložením záznamu do rodičovskej tabuľky spustia nad touto tabuľkou rozdeľovaciu funkciu. Rozdeľovacia funkcia následne miesto vloženia záznamu do rodičovskej tabuľky zabezpečí vloženie záznamu do partície.

```
CREATE TRIGGER partition_trg
BEFORE INSERT ON history
FOR EACH ROW
EXECUTE PROCEDURE trg_partition('day');
```

5.2.3 Mazacia funkcia

O mazanie starých partícií sa bude starať mazacia funkcia delete_partitions(). Využijeme pri tom kontrolnú štruktúru FOR LOOP dostupnú v jazyku pgSQL. V nej iterujeme cez všetky partície v danej schéme a zisťujeme či je dátum partície starší ako dnešný dátum.

```
FOR result IN SELECT * FROM pg_tables WHERE schemaname = 'partitions' LOOP
   table_timestamp := TO_TIMESTAMP(substring(result.tablename FROM '[0-9_]*$'),
   'YYYY_MM_DD');
   delete_before_date := date_trunc('day', NOW() - intervalToDelete);
   tablename := result.tablename;

   IF tabletype != 'month' AND tabletype != 'day' THEN
   RAISE EXCEPTION 'Please specify "month" or "day" instead of %', tabletype;
   END IF;

   IF LENGTH(substring(result.tablename FROM '[0-9_]*$')) = 10 AND tabletype =
   'month' THEN
      CONTINUE;
   ELSIF LENGTH(substring(result.tablename FROM '[0-9_]*$')) = 7 AND tabletype =
   'day' THEN
      CONTINUE;
   ELSE
   END IF;
```

Ak áno, partíciu zmažeme SQL operáciou DROP TABLE.

```
   IF table_timestamp <= delete_before_date THEN
      RAISE NOTICE 'Deleting table %', quote_ident(tablename);
      EXECUTE 'DROP TABLE ' || prefix || quote_ident(tablename) || ';';
   END IF;
```

```
END LOOP;
```

Nevýhodou tejto funkcie je, že zmaže všetky tabuľky staršie ako dnešný dátum. Nie je možné ju teda zavolať len pre jednu konkrétnu rodičovskú tabuľku.

5.2.4 Automatické mazanie partícií

Aby sme nemuseli robiť mazanie starých partícií ručne, nastavíme si v crone [24] automatické vykonávanie našej mazacej funkcie.

```
0 2 * * * root psql -U postgres zabbix -c "SELECT delete_partitions('3 days',
'day')" 2>&1 >>/tmp/zabbix_delete_partition_logs.log
```

Prvých päť znakov tohto príkazu určuje kedy sa má daný príkaz vykonať. V našom prípade je to 0 2 * * * čo znamená, že sa príkaz spustí každý deň v noci o 2:00. Ďalej sa stanoví, pod ktorým užívateľom sa tento skript vykoná. Nasleduje samotný skript kde voláme mazaciu funkciu.

Posledná časť v crone nasledujúca po skripte je presmerovanie štandardného chybového a štandardného výstupu do logovacieho súboru, aby sme si mohli overiť či sa cron úspešne spúšťa.

5.2.5 Migrácia starých záznamov do partícií

Posledným krokom je migrácia všetkých záznamov z rodičovských tabuliek do jednotlivých partícií. Tú vykonáme vybraním záznamov z rodičovskej tabuľky a opätovným vložením do tej istej tabuľky čo aktivuje trigger, ktorý následne zavolá rozdeľovaciu funkciu a tá tieto záznamy vloží do novej partície.

```
INSERT INTO history
SELECT *
FROM history;
```

5.2.6 Analýza riešenia

Popísané riešenie má nasledujúce vlastnosti:

- automaticky vytvára partície v prípade, že pri vkladaní záznamu potrebná partícia neexistuje
- nepotrebuje prázdnu databázu, je možné ju použiť na zabehnutú Zabbix inštaláciu
- jednoduchší kód ako verzia vyžadujúca prázdnu databázu
- nie sú potrebné zbytočné emergency, alebo dočasné tpl_ tabuľky, vďaka lepšiemu návrhu
- pri mazacej funkcii nie je možné mazať partície konkrétnej tabuľky

5.3 Výber optimalizovaných tabuliek

Doteraz sme sa zameriavali na to, ako budeme štrukturalizovať novú databázu. Ďalej je potrebné zvážiť, ktoré tabuľky Zabbix databázy sú vhodné na optimalizáciu. V tabuľke 5.1 môžeme vidieť počet riadkov pred optimalizáciou v jednotlivých tabuľkách.

Tabulka 5.1: Počet riadkov v tabuľkách nad 5000 záznamov

Názov tabuľky	**Počet riadkov**
history	537806592
history_uint	450477568
trends_uint	39246720
events	16986386
trends	10935209
history_str	5077260
history_text	17029
sessions	14126
items	9551
items_applications	9077
auditlog_details	5453

Vzhľadom k tomu, že partitioning sa odporúča robiť len na veľkých tabuľkách s viac ako milión záznamami, rozhodli sme sa optimalizovať prvých sedem najväčších tabuliek: history, history_uint, trends_uint, events, history_log, trends, history_log, events, trends_uint, history_uint a history.

Pre prehľadnosť partícií v databáze je tiež potrebné zvoliť ich vhodné pomenovanie. Nové partície budú teda mať nasledovný tvar:

```
<názov_rodičovskej_tabuľky>_p_<rok>_<mesiac>_<den>
```

Napríklad pre partíciu vytvorenú 21.04.2014 z history_uint tabuľky bude názov history_uint _p_2014_04_21.

Na vykresľovanie tabuliek history a trends do grafov vo webovom rozhraní Zabbix vykonáva PostgreSQL databáza SELECT operácie pri ktorých sa prehľadávajú záznamy, spĺňajúce určité kritéria. Pre rýchlejšie vyhľadávanie v záznamoch sa používa dátová štruktúra index, ktorá je použitá aj na tabuľkách history a trends. Na oboch typoch tabuliek je vytvorený kompozitný index zložený zo stĺpca itemid a clock. Rovnaký index preto použijeme aj v našich nových partíciách.

5.4 Úprava riešenia

Keďže Zabbix riešenie pre VUT sieť bolo v prevádzke už niekoľko mesiacov a databáza obsahovala veľké množstvo údajov, pri výbere riešenia zavážil najmä fakt, že prvé riešenie je možné použiť len na prázdnu databázu. Prvé riešenie tiež malo komplikovanejší kód, zbytočne používalo emergency partície a neobsahovalo žiadnu možnosť automaticky mazať

partície. Z týchto dôvodov sme sa rozhodli v prostredí VUT siete implementovať riešenie druhé, ktoré nevyžaduje prázdnu databázu. Toto riešenie sme sa následne pokúsili ešte vylepšiť a zjednodušiť.

Vybrané riešenie používa na mazanie partícií cron. Z rôznych dôvodov ako napríklad zle nastavené oprávnenia, alebo premenné prostredia však môže cron daemon zlyhať a partície nebudú zmazané. V prípade, že by nebol takýto systém dlhšiu dobu monitorovaný, môže dôjsť k prípadu, že sa zaplní disk, Zabbix nebude mať kam zapísať nové údaje a dôjde k strate dát. Snažili sme sa preto vytvoriť nezávislé riešenie, ktoré by využívalo len funkcionalitu PostgreSQL a jeho jazyka pgsql.

Aby sme sa vyhli použitiu cronu, rozhodli sme sa vytvoriť v databáze novú tabuľku partition_life. Táto tabuľka má za úlohu uchovávať informáciu o tom, ako dlho majú byť partície jednotlivých rodičovských tabuliek uložené v databáze. Tabuľka partition_life má preto nasledujúcu štruktúru:

Tabulka 5.2: Štruktúra novej tabuľky partition_life

name	**days**
events	180
history	180
history_uint	180
history_str	180
history_text	180
trends	-1
trends_uint	-1

Následne sme vytvorili novú mazaciu funkciu, ktorá bude s tabuľkou partition_life pracovať. Najskôr sme si definovali premenné:

```
CREATE OR REPLACE FUNCTION public.delete_partitions(partitionname TEXT)
 RETURNS TEXT
 LANGUAGE plpgsql
AS $function$
DECLARE
   result RECORD;
   prefix TEXT := 'partitions.';
   table_timestamp TIMESTAMP;
   delete_before_timestamp TIMESTAMP;
   v_tablename TEXT;
   v_max_partition_age INTEGER := -1;
```

Ďalej sme z partition_life získali počet dní, koľko má byť partícia ponechaná v databáze:

```
BEGIN
   v_tablename := substring(partitionname FROM '^([a-z_]*)(?=_p)');
```

```
SELECT days
INTO v_max_partition_age
FROM partition_life
WHERE name = v_tablename;
```

Pokiaľ počet dní nebol nastavený na hodnotu -1 (partície takejto tabuľky sa nebudú mazať), zmažeme dané partície konkrétnej tabuľky:

```
IF (v_max_partition_age <> -1) THEN
   delete_before_timestamp := TO_TIMESTAMP(substring(partitionname FROM
   '[0-9_]*$'), 'YYYY_MM_DD') - (v_max_partition_age * INTERVAL '1 day');
   FOR result IN SELECT * FROM pg_tables WHERE schemaname = 'partitions' AND
   pg_tables.tablename LIKE (v_tablename || '_p%') LOOP
      table_timestamp := TO_TIMESTAMP(substring(result.tablename FROM
      '[0-9_]*$'), 'YYYY_MM_DD');
      IF table_timestamp <= delete_before_timestamp THEN
         RAISE NOTICE 'Deleting table %', quote_ident(result.tablename);
         EXECUTE 'DROP TABLE ' || prefix || quote_ident(result.tablename) ||
         ';';
      END IF;
   END LOOP;
END IF;
RETURN delete_before_timestamp;
END;
```

Ako vstupný argument partitionname sme použili partíciu, od ktorej dátumu po odčítaní definovaného počtu dní v tabuľke partition_life vznikne hraničný datum, ktorý určuje, že všetky partície staršie ako tento dátum sa môžu zmazať.

Definovaná funkcia sa bude spúšťať vtedy, keď nastane pri vkladaní záznamu do partície prípad, že daná partícia ešte neexistuje a je potrebné ju vytvoriť. Najskôr sa teda zmažú partície staršie ako hodnota uchovaná v partition_life a následne bude vytvorená nová partícia pre daný deň. To zabezpečíme zavolaním funkcie delete_partitions v tele rozdeľovacej funkcie create_partition (kapitola 5.2.1):

```
EXCEPTION
WHEN undefined_table THEN

PERFORM delete_partitions(quote_ident(tablename));
startdate := EXTRACT(epoch FROM date_trunc(selector, TO_TIMESTAMP(NEW.clock)));
```

Keďže sme chceli vedieť, či nebude naše riešenie mazacej funkcie z výkonnostného hľadiska pomalšie, zmerali sme čas behu funkcie oproti starému riešeniu z kapitoly 5.2.3. Na meranie sme použili pgsql funkciu timeofday(). Obidve riešenia sme testovali počas dvoch týždňov a výsledkom bola priemerná dĺžka vykonania kódu pre starú mazaciu funkciu

17.6277 ms a pre novú 2.7372 ms.

Môžeme teda konštatovať, že oproti funkcii v kapitole 5.2.3 funkcia uvedená v tejto kapitole

- k svojej činnosti nepotrebuje cron
- maže partície len konkrétnej tabuľky, nie všetky na ktorých bola nastavený trigger pri vytváraní partícií
- urýchľuje vykonávanie navrhovaného riešenia

Kapitola 6

Zhodnotenie výsledkov testovania

Po implementácií v sieti VUT sme sa opätovne pozreli na niektoré parametre Zabbix serveru a jeho databázy a porovnali ich so stavom pred optimalizáciou. V tejto kapitole uvedieme aké výsledky boli dosiahnuté a či optimalizácia splnila ciele.

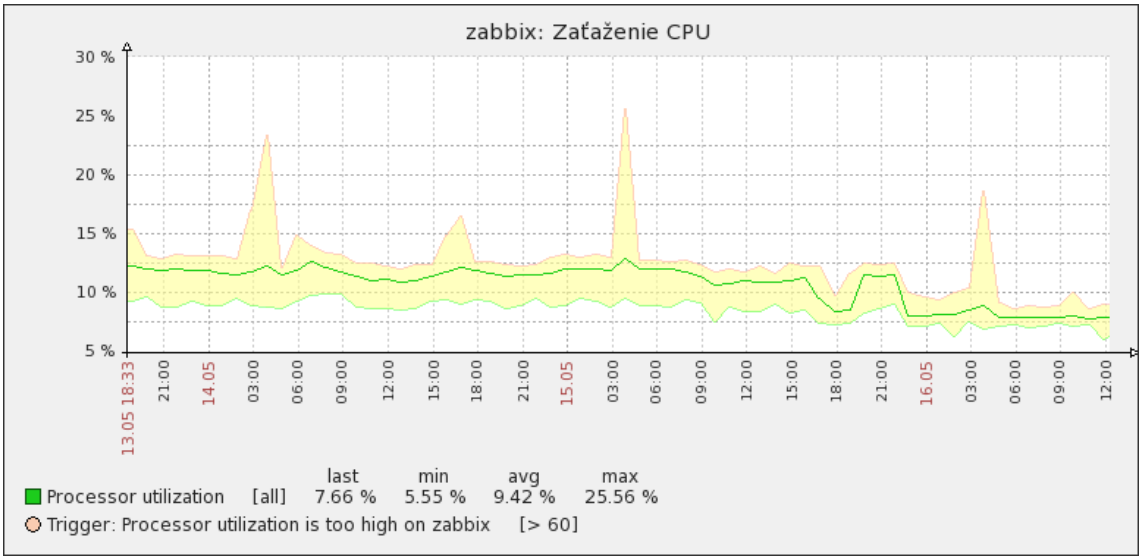
Testovanie optimalizácie trvalo celkovo dva týždne. Implementované funkcie pracovali správne, partície sa vytvárajú podľa plánu, no narazili sme aj na jeden problém. Išlo o nesprávne nastavený index v partíciách tabuľky events, ktorý bol pôvodne nastavený ako (itemid,clock). Tabuľka events však neobsahuje stĺpec itemid, obsahuje iba stĺpec eventid. Táto chyba spôsobila, že sa počas 1 dňa nevytvorili events partície. Pri pohľade na log súbor serveru Zabbix sme objavili nasledujúce chyby:

```
CONTEXT: SQL statement "CREATE INDEX events_p2014_05_10_1 on
partitions.events_p2014_05_10(itemid,clock)"
PL/pgSQL function create_partition() line 38 at EXECUTE statement
 [insert into events
 (eventid,source,object,objectid,clock,ns,value,value_changed) values
 (18965068,0,0,14676,1368233537,386595348,0,1)]
  8767:20140510:025227.091 [Z3005] query failed: [0] PGRES_FATAL_ERROR:ERROR:
  column "itemid" does not exist
```

Riešenie bolo pomerne jednoduché, stačilo opraviť rozdeľovaciu funkciu a stanoviť, že pri events tabuľke sa vytvorí kompozitný index (eventid, clock) namiesto (itemid, clock).

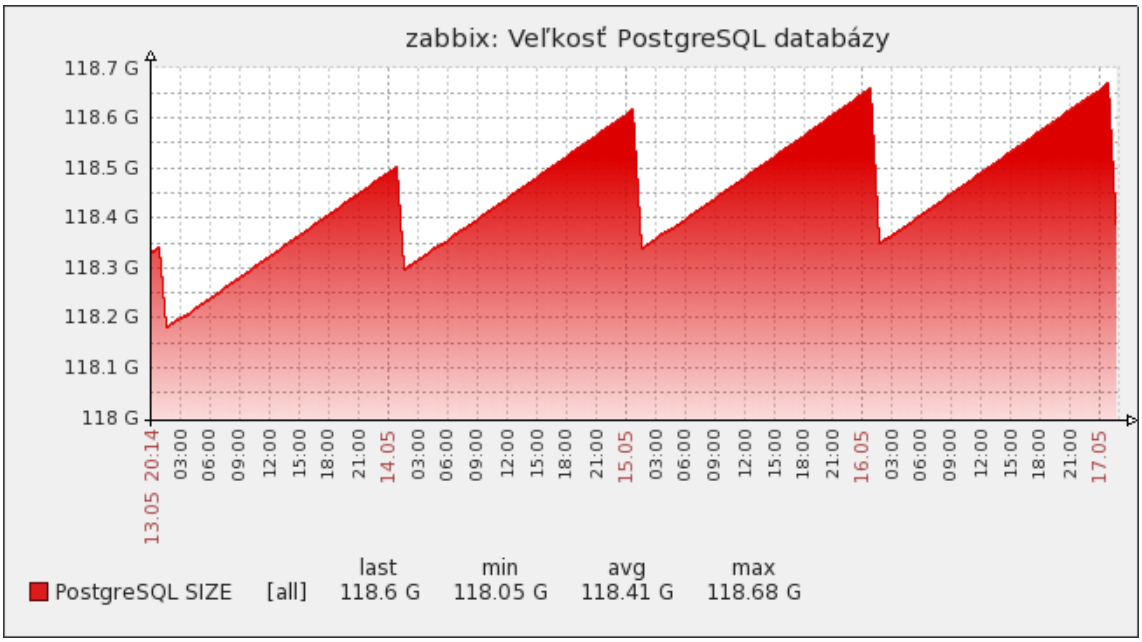
Ako prvé sme po optimalizácií skontrolovali zaťaženie procesora. Pred optimalizáciou sme na grafe 4.2 mohli vidieť periodicky sa opakujúcu zvýšenú záťaž v čase 2:39 a 13:39 spôsobenú neefektívnou operáciou DELETE. Na obrázku 6.1 zobrazujúcom zaťaženie procesora po optimalizácií vidíme, že s využitím operácie DROP na mazanie partícií už táto zvýšená záťaž neexistuje.

Problémom pred optimalizáciou bolo taktiež neustále narastanie veľkosti databázy na disku. Z obrázku 4.3 sme mohli vidieť, že za 4 dni narástla veľkosť databázy o viac ako 1 GB. Problémom bolo už spomínané používanie operácie DELETE, po ktorej sa neuvoľňovalo miesto na disku. Na obrázku 6.2 vidíme čo sa stalo po nahradení operáciou DROP. Vidíme, že po optimalizácií narástla veľkosť databázy počas 4 dní už len o 400 MB. Vždy o 2:00 v noci sa zmazali záznamy staršie ako 3 dni, čo vysvetľuje pokles veľkosti databázy na grafe v tomto čase. Napriek mazaniu starých partícií veľkosť databázy stále stúpa, čo je spôsobené



Obrázek 6.1: Zaťaženie CPU po optimalizácií

tým, že trends_* tabuľky nie sú zahrnuté pri mazaní partícií.

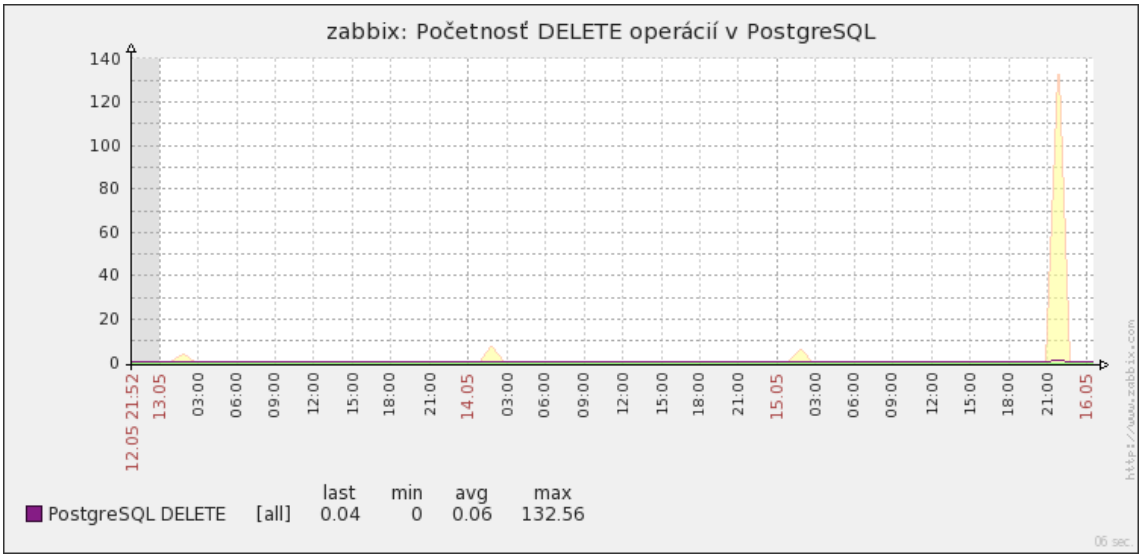


Obrázek 6.2: Veľkosť databázy po optimalizácií

K zmene logicky došlo aj v počte operácií DELETE. Ten v podstate klesol na nulu, keďže nová optimalizácia už s týmito operáciami neráta. Na obrázku 6.3 je síce vidieť jedno zvýšenie na približne 130 DELETE operácií, to je však spôsobené našim zásahom do databázy v ten deň a môžeme ho preto ignorovať.

Nové partície rozdelené po dňoch v súčasnosti po optimalizácií dosahujú veľkosť približne 1,3 milióna riadkov, čo je výrazný rozdiel oproti veľkosti najväčšej tabuľky history pred optimalizáciou, ktorá mala viac ako 500 miliónov riadkov.

K zmene tiež došlo pri plánovaní dotazov nad tabuľkami, ktoré používajú partitioning.



Obrázek 6.3: Veľkosť databázy po optimalizácií

Opäť sme vykonali dotaz z kapitoly 4. Nad tabuľkou history sme vybrali všetky riadky, ktoré splňujú podmienku, že boli vložené v dátume od 15.03.2014 do 17.03.2014:

```
EXPLAIN ANALYZE SELECT * FROM history WHERE clock > 1394841600 AND clock <
1395014400
                        QUERY PLAN
---------------------------------------------------------------
Result  (cost=0.00..64049.25 rows=2858109 width=21) (actual time=0.103..1100.390
rows=2858660 loops=1)

 ->  Append  (cost=0.00..64049.25 rows=2858109 width=21) (actual
 time=0.101..766.664 rows=2858660 loops=1)
      ->  Seq Scan on history  (cost=0.00..10.15 rows=1 width=21) (actual
      time=0.089..0.089 rows=0 loops=1)
            Filter: ((clock > 1394841600) AND (clock < 1395014400))
            Rows Removed by Filter: 410
      ->  Seq Scan on history_p2014_03_15 history  (cost=0.00..31506.61
      rows=1406096 width=21) (actual time=0.012..274.342 rows=1406361 loops=1)
            Filter: ((clock > 1394841600) AND (clock < 1395014400))
            Rows Removed by Filter: 13
      ->  Seq Scan on history_p2014_03_16 history  (cost=0.00..32532.49
      rows=1452012 width=21) (actual time=0.019..271.521 rows=1452299 loops=1)
            Filter: ((clock > 1394841600) AND (clock < 1395014400))

Total runtime: 1205.666 ms
```

Výsledkom je, že po optimalizácií plánovač počíta s CHECK obmedzeniami na jednotlivých partíciách a pri prehľadaní hľadá len v tabuľkách, ktoré majú CHECK obmedzenie v intervale určenom v našom SQL dotaze. To spôsobí, že tento dotaz vráti výsledok len za 1,2 sekundy, čo je oproti pôvodným 5,9 sekundám takmer 6 násobné zrýchlenie.

Kapitola 7

Záver

V úvodnej časti diplomovej práce sme sa zamerali na teóriu o monitorovaní počítačových sietí. Vysvetlili sme aký je rozdiel medzi aktívnym a pasívnym monitorovaním a popísali sme najpoužívanejšie monitorovacie protokoly SNMP, NetFlow, sFlow, RMON a IPMI.

Ďalšia časť práce sa venuje monitorovaciemu nástroju Zabbix, ktorý je nasadený v chrbticovej sieti VUT. Popísali sme tu ako Zabbix agent komunikuje so serverom a ako prebieha tok informácií. Uviedli sme ako Zabbix server ukladá dáta do databázy, akú štruktúru majú jednotlivé tabuľky databázy a aké veľkosti môže databáza Zabbixu dosahovať. V tejto časti sme sa tiež venovali databázovému systému PostgreSQL, ktorý je použitý v Zabbix serveri v sieti VUT. Spomenuli sme tiež funkciu Housekeeper používanú na údržbu databázy. Zhodnotili sme, že funkcia Housekeeper je vhodná len pre menšie topológie a uviedli sme aké nevýhody prináša jej používanie.

Druhá časť práce sa zameriava na praktickú časť. Porovnali sme existujúce riešenia optimalizácie databázy serveru Zabbix. Hlavnou podstatou optimalizácie je rozdelenie štruktúry rodičovských tabuliek na jednotlivé, menšie logické celky, partície. Následne sme vybrali riešenie, ktoré bolo vhodnejšie pre prostredie VUT siete. V tomto riešení sme upravili mazaciu funkciu, po ktorej došlo k urýchleniu vykonávania mazania partícií. Popísali sme tiež východzí stav Zabbix serveru a databázy pred optimalizáciou. Ďalšie kapitoly sú venované implementácií v jazyku pgSQL a zhodnoteniu testovania optimalizácie v nasadenom prostredí VUT siete.

Pri analýze funkcie Houskeeper sme uviedli dva problémy, ktoré používanie tejto funkcie prináša. Išlo o neuvoľnenie miesta na disku po vykonaní operácie DELETE a zvýšenú záťaž procesora. Navrhované riešenie rieši oba spomínané problémy, takže tento cieľ bol splnený.

Zhodnotené výsledky riešenia a krátky inštalačný návod sme sprístupnili po dohode s vedúcim práce na komunitnom fóre Zabbix a na webovej stránke http://6lab.cz, ktorá sa zameriava na bezpečnosť a správu počítačových sietí. Tento návod je tiež priložený k práci v elektronickej podobe, na CD nosiči.

Na CD nosiči je okrem inštalačného návodu možné nájsť aj zdrojové texty optimalizácie v jazyku pgSQL, zdrojový text práce a samotný text práce vo formáte PDF.

Uvedené riešenie je možné aplikovať na ľubovolnú inú sieť používajúcu Zabbix a databázový systém PostgreSQL. Z vyťaženia procesora a miesta na disku vidíme, že po optimalizácií je Zabbix server pripravený na škálovanie v sieti bez veľkého zvyšovania výkonu, alebo diskovej kapacity na Zabbix serveri.

Z pohľadu ďalšieho vývoja práce nie je veľa priestoru na vylepšovanie databázy, no v prípade, že by došlo k nadmernému nárastu počtu zariadení (z aktuálnych sto na tisíce zariadení) a ich sledovaných metrík v sieti VUT, je možné oddeliť databázový PostgreSQL

server od Zabbix serveru presunom na iný hardvér.

Pokračovanie práce by som skôr videl v rozšírení funkcionality Zabbix serveru o nové protokoly ako napr. Netflow, alebo o predpovedanie hodnôt dlhodobých trend kriviek, ktoré sú dostupné v niektorých iných monitorovacích systémoch .

Literatura

[1] CLEMM, Alexander. *Network Management Fundamentals.* Indianapolis, USA: Cisco Press, 2007. ISBN 1-58720-137-2.

[2] ANAGNOSTAKIS, K.G. Efficient Packet Monitoring for Network Management. In *Network Operations and Management Symposium.* Florence, Italy: IEEE, 2002. S. 423–436. ISBN 0-7803-7382-0.

[3] CECIL, Alisha. *A Summary of Network Traffic Monitoring and Analysis Techniques* [online]. [cit. 2. ledna 2014]. Dostupné na: <http://www.cse.wustl.edu/~jain/cse567-06/ftp/net_monitoring.pdf>.

[4] HALL, James. *Multi-layer network monitoring and analysis.* Cambridge, UK: University of Cambridge, 2003. 22-23 s. Dostupné na: <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-571.pdf>.

[5] MAURO, D. a SCHMIDT, K. *Essential SNMP, 2nd Edition.* Sebastopol, USA: O'Reilly Media Inc., 2005. ISBN 0-596-00840-6.

[6] WALDBUSSER, S., COLE, R., KALBFLEISCH, C. et al. *Introduction to the Remote Monitoring (RMON) Family of MIB Module* [online]. RFC 3577, 2003 [cit. 3. ledna 2014]. Dostupné na: <http://www.ietf.org/rfc/rfc3577.txt>.

[7] SOLIE, K. a LYNCH, L. *CCIE Practical Studies, Volume II.* Indianapolis, USA: Cisco Press, 2003. 404 s. ISBN 1-58705-072-2.

[8] CLAISE, Benoit. *Cisco Systems NetFlow Services Export Version 9* [online]. RFC 3954, 2004 [cit. 5. ledna 2014]. Dostupné na: <http://www.ietf.org/rfc/rfc3954.txt>.

[9] CISCO SYSTEMS INC. *Cisco IOS Netflow Data Sheet* [online]. 2004 [cit. 7. ledna 2014]. Dostupné na: <http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/product_data_sheet0900aecd80173f71.pdf>.

[10] SFLOW.ORG. *Traffic Monitoring using sFlow* [online]. 2003 [cit. 8. ledna 2014]. Dostupné na: <http://www.sflow.org/sFlowOverview.pdf>.

[11] PTÁČEK, J. *IPMI V2.0 zjednodušuje správu serveru* [online]. 2005 [cit. 11. ledna 2014]. Dostupné na: <http://www.svetsiti.cz/clanek.asp?cid=IPMI-V20-zjednodusuje-spravu-serveru-28112005>.

[12] OLUPS, R. *Zabbix 1.8 Network Monitoring.* Birmingham, UK: Packt Publishing Ltd., 2010. 9-10 s. ISBN 978-1-847197-68-9.

[13] *Zabbix Documentation* [online]. [cit. 13. ledna 2014]. Dostupné na: <http://www.zabbix.com/documentation>.

[14] *DB-Engines Ranking* [online]. [cit. 15. dubna 2014]. Dostupné na: <http://db-engines.com/en/ranking>.

[15] OBE, R. a HSU, L. *PostgreSQL: Up and Running.* Sebastopol, USA: O'Reilly Media, 2012. 9-11 s. ISBN 978-1-4493-2628-9.

[16] *PLpgSQL Overview* [online]. [cit. 11. března 2014]. Dostupné na: <http://www.postgresql.org/docs/9.2/static/plpgsql-overview.html>.

[17] *PostgreSQL: Inheritance* [online]. [cit. 12. dubna 2014]. Dostupné na: <http://www.postgresql.org/docs/9.2/static/ddl-inherit.html>.

[18] *PostgreSQL: Partitioning* [online]. [cit. 19. dubna 2014]. Dostupné na: <http://www.postgresql.org/docs/9.2/static/ddl-partitioning.html>.

[19] DOUGLAS, K. a DOUGLAS, S. *PostgreSQL: the comprehensive guide to building, programming, and administering PostgresSQL databases. 2nd ed.* Indianapolis, USA: Sams Pub., 2006. 155-168 s. ISBN 06-723-2756-2.

[20] *Routine Vacuuming: Recovering Disk Space* [online]. [cit. 20. března 2014]. Dostupné na: <http://www.postgresql.org/docs/9.2/interactive/routine-vacuuming.html>.

[21] *PostgreSQL: The Statistics Collector* [online]. [cit. 11. dubna 2014]. Dostupné na: <http://www.postgresql.org/docs/9.2/static/monitoring-stats.html>.

[22] *PostgreSQL Database Partitioning with Zabbix 2* [online]. [cit. 13. května 2014]. Dostupné na: <https://www.zabbix.org/wiki/Docs/howto/zabbix2_postgresql_partitioning>.

[23] *Auto Partitioning with Zabbix 2.0 and Postgresql* [online]. [cit. 17. května 2014]. Dostupné na: <https://www.zabbix.org/wiki/Docs/howto/zabbix2_postgresql_autopartitioning>.

[24] *CronHowto: Ubuntu documentation* [online]. [cit. 14. května 2014]. Dostupné na: <https://help.ubuntu.com/community/CronHowto>.

Příloha A

Seznam použitých zkratek

Tabulka A.1: Seznam zkratek s vysvětlením

Název	**Vysvětlení**
AIX	Advanced Interactive eXecutive
BIOS	Basic Input/Output System
BGP	Border Gateway Protocol
CoS	Class of Service
CPU	Central processing unit
DoS	Denial of service
DNS	Domain Name System
IETF	Internet Engineering Task Force
ICMP	Internet Control Message Protocol
IOS	Internetwork Operating System
IP	Internet Protocol
JSON	JavaScript Object Notation
MVCC	Multiversion concurrency control
MIB	Management information base
PHP	PHP: Hypertext Preprocessor
RM OSI	Reference Model Open Systems Interconnection model
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Příloha B

Obsah CD

CD má nasledující strukturu adresářů:

Tabulka B.1: Struktura adresářů CD nosiča

Adresář	**Vysvětlení**
tex	LaTeXzdrojové soubory tohoto dokumentu
pdf	text práce ve formátu pdf
src	zdrojové kódy řešení v jazyce pgSQL
manual	manuál řešení