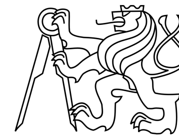
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název: Monitoring sí ových prvk
Student: Bc. Radim Dostál
Vedoucí: Ing. Tomáš Herout
Studijní program: Informatika
Studijní obor: Po íta ové systémy a sít
Katedra: Katedra po íta ových systém
Platnost zadání: Do konce letního semestru 2017/18

Pokyny pro vypracování

Navrhn te a realizujte systém zabývající se sb rem, vyhodnocením a vizualizací dat ze sí ových prvk . Hlavní d raz je kladen na minimální zát ž monitorovaných prvk , vysoký po et sledovaných veli in a škálovatelnost.

Komponenta sbírající data (kolektor) využije pro získání dat protokol SNMP. Použijte tzv. "bulk" dotaz za ú elem minimalizování zát že. Nasbíraná data p edejte pomocí AMQP zpráv k vyhodnocení a vizualizaci.

Komponenta vyhodnocující data si p ijaté zprávy uloží do in-memory databáze za ú elem porovnání zm n aktuální hodnoty oproti p edchozím. Samotné vyhodnocení dat prob hne oproti definovatelným formulím nad množinou dat v in-memory databázi a na základ výsledku bude této veli in p i azena klasifikace závažnosti. V p ípad zm ny v klasifikaci dojde k notifikaci o této skute nosti.

Pro poslední komponentu, vizualizaci dat, využijte již existující ešení vhodné pro tento ú el a p edejte mu data odeslaná z kolektoru.

ešení otestujte a zm te výkonnost.

Seznam odborné literatury

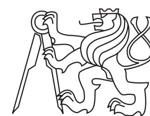
Dodá vedoucí práce.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 20. ledna 2017

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



Diplomová práce

Monitoring síťových prvků

Bc. Radim Dostál

Vedoucí práce: Ing. Tomáš Herout

8. května 2017

Poděkování

Na tomto místě bych rád poděkoval vedoucímu práce Ing. Tomášovi Heroutovi, který měl vždy zásobu konstruktivní kritiky, která mi při vývoji velmi pomohla a také se nebál nechat několik kritických rozhodnutí pouze na mě. Dále bych rád poděkoval společnosti TETA s.r.o. a výslovně jejímu řediteli Ing. Zdeňku Smržovi za poskytnutí testovacího prostředí pro vývoj.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 8. května 2017 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Radim Dostál. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Dostál, Radim. *Monitoring síťových prvků.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato diplomová práce se zabývá implementací nástroje, který slouží pro dohledování síťových zařízení prostřednictvím SNMP. Nástroj je rozdělen do tří samostatných komponent, které spolupracují a řeší dílčí problémy. První komponenta data získává a předává je dalším komponentám ke zpracování. Druhá komponenta data ukládá a používá je pro generování grafů. Třetí komponenta data vyhodnocuje pomocí předepsaných výrazů a dle výsledků informuje uživatele.

Klíčová slova dohled, rrd, whisper, snmp, graf, zabbix, grafana, graphite

Abstract

The thesis deals with implementation of tool which serves for monitoring a network devices through SNMP. This tool is divided to three separate components which cooparate together and solves partial problems. First component obtains data and provides them to another components. Second one saves data to persistent storage and generates graphs based on that data. Third component evaluates expressions with data and notify users about the results.

Keywords monitoring, rrd, whisper, snmp, graph, zabbix, grafana, graphite

Obsah

Úvod **1**

1 Cíle práce **3**

2 Analýza **5**
2.1 Časové řady 5
2.2 Komunikace 11
2.3 Graphite 13
2.4 Grafana 18
2.5 Management information base 19
2.6 Simple Network Management Protocol 19
2.7 Zabbix 25

3 Návrh **29**
3.1 Kolektor 29
3.2 Ukládání dat a generování grafů 32
3.3 Vyhodnocování dat 32

4 Realizace **37**
4.1 Kolektor 37
4.2 Ukládání dat a generování grafů 45
4.3 Vyhodnocování dat 46

5 Testování **51**
5.1 Testovací prostředí 51
5.2 Kolektor 51
5.3 Generování grafů a ukládání dat 53
5.4 Vyhodnocování dat 54

Závěr **57**
Budoucí vývoj 58

Literatura **59**

A Seznam použitých zkratek **63**

B Další přílohy **65**

C Obsah přiloženého CD **69**

Seznam obrázků

2.1 AMQP Architektura 12
2.2 Graphite Architektura 14
2.3 Graf vygenerovaný pomocí Graphite Web 15
2.4 Graf vygenerovaný nástrojem Grafana 18
2.5 Ukázka SNMPv3 entity 23
2.6 Ukázka formátu zprávy SNMPv3 23
2.7 VACM vyhodnocení přístupu 24

3.1 Databáze Kolektoru 31
3.2 Databáze komponenty vyhodnocující data 34

4.1 Architektura Kolektoru 37
4.2 Vztahy nejdůležitějších datových struktur 43
4.3 Architektura komponenty vyhodnocující data 47

5.1 Doba běhu v závislosti na počtu paralelních vláken 52
5.2 Zrychlení na základě počtu paralelních vláken 52
5.3 Využití procesů pro I/O operace 54
5.4 Podíl chybujících triggerů a doby běhu s 10 vlákny 55
5.5 Závislost doby běhu na počtu výpočetních vláken s 5 % chybujících triggerů 56

Úvod

Problém dohledování síťových zařízení je běžná záležitost, která je u kvalitního správce řešena denně. Ze zkušenosti vím, že většina přednášek, které jsou součástí veřejných akcí a týkají se dohledových služeb, je hojně navštěvována a po skončení se vedou dlouhé diskuze. Protože jsem sám byl několikrát součástí podobných diskuzí a postaven před problémy, které nebylo zkrátka lehké vyřešit prostřednictvím existujících možností, rozhodl jsem se přispět do množiny těchto nástrojů prostřednictvím této práce v níž se specializuji na dohledování síťových zařízení prostřednictvím SNMP. V dnešní době existuje velké množství nástrojů podobných možností, ale protože je stále mnoho přístupů, které nejsou zcela využity, pokusím se tak přispět.

Tato implementace přistupuje k problému jinak než většina současných řešení. Je rozdělena do několika komponent, které spolu kooperují a poskytují tak požadovanou funkcionalitu. Modularita je jedna z vlastností chybějící u většiny ostatních nástrojů. Díky tomu je nástroj relativně flexibilní a do budoucna dobře udržitelný za předpokladu, že je možné některé komponenty vyměnit. Další obecnější změna v přístupu k dohledování je využití databáze v operační paměti pro vyhodnocení výrazů generujících problémové stavy. Běžné dohledové systémy často využívají relační databáze, které pro tento účel nejsou příliš vhodné a při větší zátěži nejsou schopné podávat požadované výkony. Protože se ve většině případů nástroje řadí do kategorií, které jsou rozděleny na nástroje generující grafy z dat a na nástroje, které data vyhodnocují pomocí výrazů. Cílem tedy je využít získaná data pro obě problematiky a nesnažit se je získat znovu.

Kapitola 1

Cíle práce

Cíle této práce vycházejí z několikaletého pozorování situace v prostředí středně velké síťové infrastruktury a její správy. Tato infrastruktura obsahuje stovky až jednotky tisíc síťově aktivních zařízení, které musí být aktivní 24 hodin denně a to 7 dní v týdnu. Abychom jako správci síťové infrastruktury mohli poskytnout, co možná nejlepší výsledky z pohledu bezproblémového provozu, tak musíme provozovat relativně velké množství nástrojů, které nám s tím pomohou. Provoz většího množství nástrojů je vždy problematický a bylo by výhodou mít například pro dohled a grafy jeden komplexnější nástroj. Tato diplomová práce si klade za cíl udělat první krok k vyřešení tohoto problému způsobem, který by byl co možná nejvhodnější pro zobecnění a pro přenos do jiného prostředí. To je jeden z hlavních důvodů, proč je rozdělena do komponent, které spolu komunikují univerzálním komunikačním kanálem. Jak specifikuje zadání práce, není nutné implementovat veškeré komponenty od začátku a jedna z možností je tedy využití už existujícího řešení pro generování grafů. Jak jsem zmínil, tak výsledný produkt musí být obecný pro další použití, a protože se očekává růst infrastruktury, tak i v rámci možností dobře škálovatelný a výkonný.

Jádrem této práce bude komponenta nazývaná Kolektor, která se bude starat o získávání dat a jejich předávání k dalšímu zpracování. V dnešní době je pravděpodobně nejrozšířenějším protokolem pro poskytnutí dat v tomto měřítku ze strany dohledovaného zařízení protokol SNMP. Tento protokol implementuje většina zařízení, které je možné připojit do sítě provozované nad rodinou protokolů TCP/IP. Kolektor by také měl poskytnout možnost škálování a to v podobě paralelního získávání dat. U méně výkonného zařízení může dojít k jeho několika sekundovému zahlcení při zpracování velkého množství SNMP dotazů a generování odpovědí. Proto je nutné implementovat možnost tzv. bulk dotazů, které poskytují získání většího množství dat pomocí jednoho dotazu. Již zmíněné poskytnutí získaných dat bude provedeno, co možná nejuniverzálnější cestou.

Pro každého správce je jistě důležité mít přehled o stavu sítě, což zahrnuje například zatížení komunikujících rozhraní, která data přenášejí, a obdobných informací. Tento přehled získáme nejlépe prostřednictvím grafů, které tyto informace poskytují v kvalitním a přehledném provedení. Grafy využívá většina nástrojů, které pomáhají se správou sítí, a ve většině případů jsou jejich možnosti dostatečné, ale pokud požadujeme komplexnější řešení, které nám data poskytne, například i pro jiné operace, tak v mnoha případech narazíme na

nepřekonatelnou bariéru. Dalším častým nedostatkem těchto nástrojů je nedostatečný výkon z pohledu získání dat, uložení dat a nebo v obou případech. To je většinou zapříčiněno několika důvody a to například nevhodně zvoleným typem databáze nebo špatně navrženým uživatelským rozhraním. Naproti tomu je několik nástrojů, které se vydaly cestou řešení jen dílčího problému, jako je například jen uložení dat a generování grafů. Tyto nástroje se o získávání dat nestarají. To je přístup, který můžeme nazvat modulárním, který se v historii už několikrát ukázal jako velmi výhodný, zvláště pak v otázkách výkonu a budoucího vývoje. Modulární přístup bych chtěl zachovat i v rámci mé práce a postupovat tak, že každá komponenta je potencionálně nahraditelná. Protože téměř každá společnost využívá svůj systém pro správu informací, tak se dá předpokládat, že další důležitou vlastností bude rozhraní pro programové získání dat či grafů tzv. API. Tato vlastnost již bohužel není zdaleka standardem mezi dostupnými nástroji, a proto se jí pokusím z části implementovat v této práci.

Další komponenta se zabývá vyhodnocením výrazů pomocí obdržených dat a reakcí na ně, nikoliv jejich získáváním, čímž dodržíme vlastnost modularity. Tato data by měla přijímat nejlépe univerzální cestou, která pro tento účel dává smysl. Vyhodnocení výrazů bude probíhat paralelně. Stejně jako v části generující grafy bude nutné poskytnout výsledky vyhodnocených výrazů pro další zpracování, a to jak v podobě přehledu, tak v podobě notifikací. Notifikace se budou předávat prostřednictvím obecného kanálu, který bude možné dále využít pro zpracování notifikací, ale to není předmětem mé práce. Důležitou vlastností této komponenty bude výkon, protože to je často zásadní problém běžně dostupných řešení. Protože je relativně velká škála nástrojů, řešících obdobný problém, tak se budu jedním z nich inspirovat při návrhu.

1. Kolektor
   a) SNMP s použitím bulk dotazů
   b) paralelní zpracování
   c) poskytnutí dat univerzální cestou
2. Komponenta generující grafy
   a) škálovatelné ukládání dat pro grafy
   b) API poskytující data i obrázky
   c) univerzální přístup a žádná závislost na hodnoty generované do grafu
3. Komponenta vyhodnocující data
   a) paralelní zpracování
   b) notifikace pomocí obecného kanálu
   c) výkon pomocí databáze v operační paměti

Kapitola 2

Analýza

V této kapitole je hlavním cílem představit existující řešení a nástroje, které budou následně použity při implementaci a nasazení celého komplexního řešení. Protože většina těchto nástrojů nemá ve svém relativně malém okruhu co se možností týče konkurenci, nebudu je tedy až na výjimky konfrontovat s jinými. Například databáze Whisper použita sadou nástrojů Graphite vychází mimo jiné ze staršího RRD, které zde pro úplné pochopení přiblížím, hlavně proto, že většina změn provedena při jeho vývoji vychází z nedostatků RRD. Také zde popíši některé obecně známé postupy pro ukládání časových řad. Popíši zde i dohledový systém Zabbix, kterým jsem se inspiroval při návrhu architektury části vyhodnocující data a budu používat i jeho terminologii. Protože je práce založena na získávání dat pomocí protokolu SNMP, tak zde popíši všechny jeho verze a jejich vlastnosti.

2.1 Časové řady

Tato data jsou známá pro své využití v oblasti statistiky [1], kde vznikají při pozorování událostí v čase (počasí nebo například ekonomické údaje). Vznikají také například při pozorování Stochastického procesu [2, s. 4], který generuje hodnoty závislé na jiné hodnotě, která se ve většině případů interpretuje jako čas. Hodnotou, kterou nám dává tento proces, rozumíme náhodnou veličinu. V našem případě lze chápat zdroje dat, tedy například konkrétní počítadlo, které svou hodnotu v čase zvětšuje o konstantní přírůstek. Data jsou jednoznačně uspořádaná na základě času.

Tato práce se zabývá dohledem, který je velmi závislý na datech časové řady. Snadno si lze představit, že veškerá data získaná a využívána pro dohled jsou data závislá na čase. Vždy nás zajímá například zatížení procesoru v čase „t“ nebo hodnota počítadla v konkrétních intervalech. Protože předem známe strukturu dat, tak lze navrhnout jejich efektivní ukládání a zpracování. Díky tomuto předpokladu vzniklo několik projektů, které se zabývají tímto problémem. Nejdůležitější z nich představím v následující části textu. Na čas těchto dat se lze dívat jako na diskrétní časové značky, data si v zjednodušené formě lze představit jako páry jednotlivých hodnot (získaná hodnota a časová značka). Jsou ve většině případů seřazena dle času, což je výhodné i pro většinu dohledových systémů. Pro dohledový systém ve většině případů není vhodné udržovat velké množství historických dat, protože databáze dosahují relativně

velkých velikostí a vyhodnocování předepsaných výrazů nad nimi je neefektivní a často nedává smysl. Pro vyhodnocení většiny výrazů je dostačující uchovávat pouze jednotkové množství historických dat.

Naopak pro systém, který se zabývá generováním grafů, je ve většině případů důležité uchovávat, co možná nejvíce historických dat. Uživatel téměř vždy požaduje vidět historický vývoj hodnot, což pro velké množství sbíraných dat a jejich rychlé zpracování (vykreslování) může být obtížné. Ale protože známe povahu dat, je možné udělat několik výhodných kroků, které nám uložení dat zjednoduší za cenu ztráty přesnosti, ale s touto ztrátou jsme schopni se ve většině případů smířit. Hlavním krokem je tzv. konsolidace dat, tedy jejich upravení do menšího množství dat na základě předem určené funkce. Máme tedy k dispozici funkce, které nám spočtou a uloží například hodinové průměry, týdenní maxima a tak dále. Tyto vlastnosti jsou jedním z hlavních bodů databází, které se zabývají efektivním ukládáním dat pro generování grafů. Tato vlastnost se uživateli projeví tak, že (samozřejmě dle nastavení) si může zobrazit například posledních několik týdnů až na úroveň hodnot po několika minutových intervalech, ale například několik měsíců zpět v čase si lze zobrazit jen na úroveň průměrných hodnot za den. Další výhodnou vlastností těchto dat je, že po nastavení konsolidace a retence může systém předem vypočítat velikost databáze a ještě před ukládáním hodnot určit, zda má dostatečný datový prostor.

2.1.1 Round Robin Database

Databáze RRD byla jedna z prvních, která se zabývala ukládáním časových řad. Historie této databáze sahá až do roku 1999, kdy vývojář Tobias Oetiker ze Švýcarska začal s jejím vývojem [3]. Vývoj je stále živý a postupem času přicházejí i nové funkcionality. Projekt zahrnující tuto databázi se jmenuje souhrnně RRDTool. Již název napovídá, že se projekt také zabýval otázkou, jak data reprezentovat, měnit a dále zpracovávat. Protože tato databáze inspirovala většinu projektů zabývajících se ukládáním tohoto typu dat, tak ji zde představím detailněji. V dnešní době se s touto databází setkal asi každý, kdo se kdy zabýval dohledem a generováním grafů dat závislých na čase. Takže není překvapením, že pro komunikaci s těmito nástroji vznikla celá řada knihoven a skriptů, téměř pro každý dnes běžně používaný programovací jazyk. Obecně lze tuto databázi chápat jako statické kruhové pole, které je uložené v souboru. Jak jsem naznačil, tak velikost souboru je známa již při jeho vytvoření. Databáze je reprezentována pomocí souborů, které mohou obsahovat až několik kruhových polí nazývaných RRA.

2.1.1.1 Vytvoření databáze

Pro vytvoření databáze slouží příkaz „create“, který během svého dlouholetého vývoje získal možnost komplexního zápisu své konfigurace, která by měla být každému administrátorovi více než blízká. Díky této konfiguraci je jednoduché vytvořit databázi, která dále spravuje ukládání dat, poskytuje jejich retence a konsolidace. Následující příklad konfigurace podrobněji popíši [4, kap. rrdcreate].

```
1  rrdtool create filename.rrd \
2          --step 300 \
```

```
3            DS:inErrors:COUNTER:600:0:4294967296 \
4            DS:outErrors:COUNTER:600:0:4294967296 \
5            RRA:AVERAGE:0.5:1:576 \
6            RRA:AVERAGE:0.5:6:672 \
7            RRA:AVERAGE:0.5:24:372 \
8            RRA:AVERAGE:0.5:288:730 \
9            RRA:MAX:0.5:6:672 \
10           RRA:MAX:0.5:24:372 \
11           RRA:MAX:0.5:288:730 \
12           RRA:MIN:0.5:6:672 \
13           RRA:MIN:0.5:24:372 \
14           RRA:MIN:0.5:288:730
```

Listing 2.1: Příklad vytvoření RRD

Příkaz pro vytvoření databáze lze rozdělit do 3 základních částí. První částí jsou obecné parametry příkazu „create“, které nastavují chování databáze. Nejdůležitějším parametrem je „step“, který určuje interval příchodu nových hodnot a argument určující název souboru. Dále následují argumenty samotné databáze, obecně ve tvaru:

`DS:název:datový typ:heartbeat:minimální hodnota:maximální hodnota`

DS klíčové slovo reprezentující, že jde o řádek definující „Data source“

název název datového zdroje, na který se odvoláváme při generování grafů

datový typ datový typ, který může být GAUGE, COUNTER, ABSOLUTE, COMPUTE, více podrobností lze dohledat ve zdroji [4]

heartbeat parametr určující dobu čekání na novou hodnotu než se uloží hodnota jako neznámá – z pravidla nastaven na hodnotu 1,5 násobku parametru „step“

minimální a maximální hodnota
aby databáze mohla předem určit velikost souboru, musíme znát i rozsah ukládaných hodnot, což nastavíme těmito parametry

Datových zdrojů lze pro jednu databázi použít několik a pro každý se vytvoří všechny nastavené archivy (RRA). Databáze podporuje i složitější typy, jejíchž hodnoty se určí na základě ostatních datových zdrojů.

`RRA:konsolidační funkce:XFF:počet dat:počet řádků`

RRA Round Robin Archive neboli kruhové pole, které se chová na základě následujících parametrů

konsolidační funkce
jedna z dostupných konsolidačních funkcí – AVERAGE, MIN, MAX, LAST

XFF „xFileFactor“ je parametr určující přípustné množství neznámých hodnot pro vypočítání konsolidační funkce, v tomto případě 50 % hodnot může být neznámá

počet dat určuje počet hodnot, ze kterých je následně vypočítána konsolidace, pro hodnotu jedna nepoužije konsolidaci viz 5. řádek v příkladu 2.1

počet řádků počet řádků archivu, tedy počet historických hodnot, které bude archiv udržovat

Například parametr *RRA:MIN:0.5:24:372* nastaví archiv tak, aby použil konsolidační funkci MIN na 24 přijatých hodnot a aby zkonsolidovaných hodnot udržoval 372, takže tento archiv bude udržovat 31 dní dvouhodinová minima.

2.1.1.2 Přidání dat

Klíčová funkce, jako přidání dat do databáze, je jednoduchá a efektivní. Její argumenty, jak by čtenář jistě předpokládal, nejsou komplikované. Je jen důležité zmínit, že pokud databáze obsahuje více datových zdrojů, je důležité jejich hodnoty vložit v pořadí, v jakém jsou definované v databázi [4, kap. rrdupdate].

```
rrdtool update název_souboru.rrd časová značka:hodnota...
```

časová značka pro přesnost dat databáze poskytuje možnost vložení konkrétního času, buď formou časové značky, nebo parametrem „N“, který je nahrazen za aktuální čas

hodnota hodnota pro uložení nebo znak „U“, který reprezentuje neznámou hodnotu

2.1.1.3 Generování grafu

Jednou z velkých výhod RRD je možnost generování grafů pomocí dostupného skriptu, který využívá valná většina nástrojů. Tyto grafy lze vygenerovat v několika různých formátech, jako jsou PNG, SVG, a nebo je jednoduše nechat vygenerovat na standardní výstup v binární podobě. Tento příkaz lze rozdělit do několika částí. V první části definujeme soubory a příslušné datové zdroje z nich, další část obsahuje úpravu dat před vykreslením, následující část vykreslí data do grafu a poslední část vygeneruje legendu.

```
1  rrdtool graph --slope-mode --start=end-1w \
2    --title="Titulek grafu" \
3    --vertical-label="popisek osy y" \
4    'DEF:ds1=soubor1.rrd:nazevDs1:AVERAGE' \
5    'DEF:ds2=soubor2.rrd:nazevDs2:MAX' \
6    'CDEF:ds1b=ds1,8,*' \
7    'CDEF:ds2b=ds2,8,*' \
8    'AREA:ds1b:Popisek ds1' \
9    'LINE:ds2b:Popisek ds2' \
10   'GPRINT:ds1b:AVERAGE:Ds1 AVG %3.1lf %s' \
11   'GPRINT:ds2b:MAX:Ds2 MAX %3.1lf %s'
```

Listing 2.2: Příklad vytvoření RRD

V první části je několik parametrů, které určují obecné nastavení grafu. Další parametry určují popisky a název. Pro účely této práce není nutné popisovat další podrobnosti o vykreslování dat. Více informací lze dohledat v [4, kap. rrdgraph].

2.1.2 Whisper

Databáze Whisper byla poprvé vydána v roce 2011 a její vývoj byl zahájen z důvodu doplnění nedostatků RRD, která nepodporuje dostatečnou úroveň škálovatelnosti. Veřejnosti je dostupná pod licencí Apache License 2.0. Specializuje se jen na ukládání číselných hodnot s časovou značkou a mezi hodnotami nehledí na rozdíly, což znamená, že o úpravu hodnot, které jsou například typu počítadlo, se musí postarat procesor generující grafy. Vlastnost konstantní velikosti databáze zůstává.

Tato databáze se opět skládá ze souborů, které jsou vytvořeny pro každý datový zdroj. Tím ztrácíme možnost mít více datových zdrojů v jednom souboru, což je cena za větší škálovatelnost. Každý soubor stejně jako v RRD obsahuje několik archivů, které určují jak dlouho a v jaké přesnosti budou data perzistentně uložena. Jedno z vylepšení je větší srozumitelnost konfigurace databáze, viz následující příklad:

```
whisper-create --xFilesFactor=0.5 --aggregationMethod=average
    název_souboru.wsp 300:576 30m:14d
```

Význam parametru *xFilesFactor* známe z RRD, poté následuje název souboru s příslušnou příponou a dále retenční parametry. Druhý z nich říká, že v databázi chceme uchovávat půlhodinové průměry po dobu 14 dní. V souboru tak vznikne archiv s touto vlastností. V RRD jsme měli možnost ukládat data pomocí různých agregačních funkcí v rámci jednoho souboru, ale v této databázi je můžeme ukládat jen pomocí jedné. Takže použití dalších agregačních funkcí a metod spadá na stranu vykreslování grafů. Pokud chceme ukládat data s více různými agregacemi, jako tomu je v RRD, tak musíme vytvořit více souborů pro stejná data s jinou agregační funkcí. Pro práci s databází získáme kromě skriptu „whisper-create“ několik dalších naprogramovaných v jazyku Python, které ve zkratce popíši:

rrd2whisper jak název napovídá, jedná se o jednoduchý skript pro migraci RRD do databáze Whisper – i to nám ukazuje jejich úzkou vazbu

whisper-update skript pro přidání dat do databáze

whisper-resize umožňuje změnu retence dat existujícího souboru s přepočítáním již uložených dat

whisper-dump výpis metadat databáze – retence, agregační funkce a další provozní informace

whisper-fetch vypíše data uložená v databázovém souboru na standardní výstup pro další zpracování, nabízí možnost zvolení jednoho ze dvou výstupních formátu dat, a to „pretty“, který je čitelný pro administrátora, a „JSON“ pro další zpracování

whisper-merge sloučí několik existujících souborů do jednoho

whisper-info vypíše nastavení archivů a retence dat

Při získávání dat z databáze je použitý první archiv, který vyhovuje zvolené časové periodě. Například když ukládám hodnoty z jednoho dne v nezměněné podobě a chci vykreslit graf s hodnotami z tohoto dne, tak získám data neovlivněná konsolidací. Ale pokud chci vykreslit graf s daty z předešlého dne, tak získám data konsolidací ovlivněná, která pochází z archivu s delší retencí [5].

2.1.3 RRD vs. Whisper

Některé rozdíly těchto dvou databází jsem již zmínil v části o databázi Whisper, ale je ještě několik dalších, které zmíním zde. Jeden z důvodů, kvůli kterému vznikla databáze Whisper, je nemožnost v RRD přidání hodnoty před poslední uloženou hodnotu. Takže při ukládání více hodnot, v rámci jednoho zápisu, je nutné data zapisovat v konkrétním pořadí. V době vzniku databáze Whisper, RRD nepodporovala hromadné zápisy, což je další kritická vlastnost pro pohodlné škálování. Na druhou stranu databáze Whisper nechává většinu důležité práce na uživateli a stará se jen o uložení dat. K tomu lze přičíst generování grafů, které je nutné generovat externě na základě získaných dat, nebo přijímání dat s jiným intervalem, než který je nastaven v archivu. RRD poskytuje pro tato data dočasnou paměť a až po uplynutí předem nastaveného intervalu data uloží, anebo s nimi provede konsolidační funkci a poté je uloží.

Výkon databáze Whisper je častým obsahem mnoha diskuzí. Její tvůrce popisuje, že provedl veškeré optimalizace, aby se co možná nejvíce přiblížil výkonu RRD, ale je nutné vzít v úvahu, že Whisper je databáze naprogramována v jazyku Python a je vyvíjena pouhých 6 let, za to RRD je naprogramována v jazyku C a má za sebou 18 let vývoje. Autor tvrdí, že databáze je zhruba 2× až 3× pomalejší než RRD pro operaci uložení a 2× až 5× pomalejší pro ostatní operace [6].

Dalším rozdílem těchto databázových systémů je jejich přístup k zařazení právě přijaté hodnoty. Protože jsou to specializované systémy pro uložení časových řad, tak neukládají data s časem, tak jak ho přijmou, ale šetří prostor úložiště pomocí předem spočítaných intervalů (při vytváření vkládáme parametr „step“). Potom je relativně složité odhadnout, ke kterému časovému bodu patří která přijatá hodnota, protože se samozřejmě může stát situace, kdy jsou přijatá data uprostřed intervalu. Databáze Whisper se s tímto problémem vypořádá tak, že čas přijatých dat zarovná na první nižší prázdný datový slot a uloží je. Z čehož plyne, že ve chvíli, kdy se jedna hodnota ztratí a další přijde o několik sekund dříve, tak máme zkreslená data, která jsou však v rámci našeho přednastaveného intervalu. Takže existují situace, kdy nám tato vlastnost nevadí. Navíc pokud data chodí častěji než je uživatelem nastavený interval, tak v databázi nakonec zůstanou jen poslední přijaté hodnoty v rámci intervalu. Na rozdíl od tohoto přístupu má RRD možnost uložení přijatých dat do dočasné paměti a použít je až bude nutné vypočítat novou hodnotu pro uložení. RRD celý interval sbírá data, která ukládá do zmíněné dočasné paměti, obohacuje je o své interní výpočty a po uplynutí intervalu z nich vygeneruje následující hodnotu pro uložení do databáze. Takže v prostředí, kde je kladen větší důraz na přesnost dat, je to přednost. Toto je jeden z hlavních důvodů proč RRD neu-

možňuje aktualizovat data v minulosti, tedy doplnit mezery dat [7]. Podrobněji jsou přístupy ukládání dat tohoto typu popsány ve zdroji [8].

2.2 Komunikace

2.2.1 Advance Message Queueing Protocol

Zkráceně AMQP je protokol, který umožňuje rozdílným aplikacím komunikovat většinou prostřednictvím mezilehlého zařízení, které se nazývá zprostředkovatel. Zprostředkovatel je aplikace, která implementuje tento protokol, především jeho specifikovanou architekturu, která je znázorněna na obrázku 2.1. Protokol je navržený jako velmi obecná platforma, je tedy použitelný ve většině případů komunikace. Popisuje jak samotná komunikace vypadá na síťové úrovni a sémantiku dat, která si aplikace vyměňují. Specifikace explicitně popisuje sémantiku protokolu na straně serveru, takže každá implementace je totožná. Hlavním důvodem vzniku protokolu byla možnost propojení vzájemně odlišných systémů. Protokol umožňuje spolehlivé doručování zpráv právě díky možnostem zprostředkovatele, který je uchová, dokud nebudou vyzvednuty příjemcem [9]. Hlavní tři komponenty, které umožňují komunikaci pro dosažení této funkcionality, jsou popsány v následující části [10]. Zprostředkovatel umožňuje například využití jednoduchého směrování založeného na textovém řetězci, tento řetězec je součástí přijaté zprávy.

binding vztah mezi komponentou exchange a frontou definující směrovací pravidla

exchange komponenta, která přijímá zprávy a směruje je do příslušných front na základě pravidel, která definuje „binding“ (propojení exchange a fronty), exchange může být několika typů:

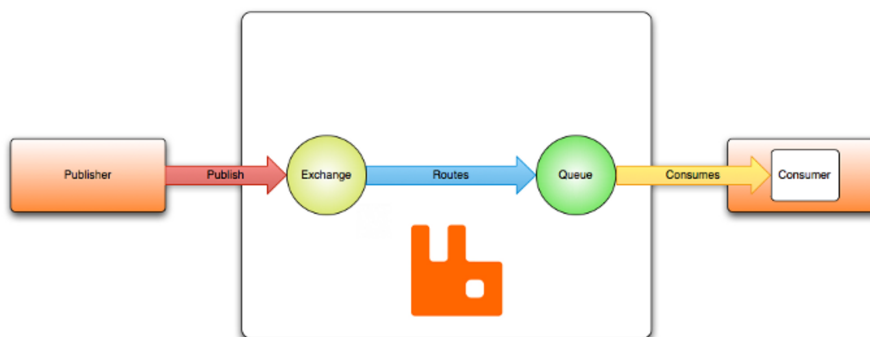
- **direct** na základě směrovacího řetězce odešle zprávu do příslušné fronty
- **fanout** ignoruje směrovací pravidla a přepošle zprávu do všech front, které jsou připojeny
- **topic** na základě směrovacích pravidel, které mohou být komplexnější a může jich být více než v případě typu direct, odešle zprávy do příslušných front
- **headers** už název napovídá, že pro směrování používá parametry hlavičky AMQP

message queue fronta pro zprávy, ze které jsou klienty zpracovávány

Protokol samozřejmě umožňuje veškeré základní zabezpečovací prostředky, jako je šifrování pomocí asynchronní šifry a ověření komunikující strany. K dispozici je několik implementací tohoto protokolu.

2.2.2 RabbitMq

Nástroj o kterém jeho vývojáři tvrdí, že je to nejvíce nasazovaný prostředník pro zasílání zpráv [12]. Je to nástroj, který se řadí do skupiny tzv. middleware, které zprostředkovávají komunikaci nebo další parametry při propojení



Obrázek 2.1: AMQP Architektura [11]

více systémů do jednoho komplexního celku. Protože i tato diplomová práce propojuje více systémů, tak ho ve zkratce představím. Tento nástroj vznikl před 10 lety a má za sebou dlouholetou tradici a snahu zdokonalování. Je používán ve velkém množství prostředí, jako jsou například společnosti Cisco Systems, Instagram, Ford a mnoho dalších. Jeho hlavní vlastností je implementace AMQP, který mu napomáhá zprostředkovat asynchronní zasílání zpráv, poskytnutí fronty pro zprávy a dalších možností, jako potvrzování přijetí zpráv a tak dále. V dnešní době, kdy se řeší možnosti distribuovaných systémů, jistě obstojí, protože poskytuje možnost nasazení více instancí v tzv. „clusteru“. Tento „cluster“ představuje spuštění více instancí této služby většinou v jiných lokalitách a poskytuje jejich spolupráci. Poskytuje samozřejmě mnoho dalších možností, které jsou nad rámec této diplomové práce, takže se s nimi nebudu zabývat.

2.2.3 Java Message Service

Obdoba funkcionality, kterou poskytuje AMQP, je standard popisující komunikaci JMS. Popisuje API a další části, které jsou určeny pro výměnu zpráv mezi aplikacemi naprogramovanými v jazyku Java. Tento middleware poskytuje především funkcionalitu zasílání asynchronních zpráv, které prochází frontami nebo obdobnými částmi implementovaného zprostředkovatele. Zařizuje především spolehlivou komunikaci a jednotný tvar rozhraní [13].

2.2.4 ActiveMq

Obdobně jako zprostředkovatel RabbitMq, je ActiveMq mimo jiné implementací JMS. Je poskytován pod licencí Apache 2.0 licence. Protože je jeho využití v dnešní době široké, má kvalitní dokumentaci a podporu dalších nástrojů, jako jsou například frameworky pro práci s ním. Podporuje celou řadu dalších vlastností, jako je například škálování, zmíněnou asynchronní komunikaci. Při práci s ním není nutné znát informace o koncových stanicích, které ho využívají. Poskytuje REST API a existuje k němu webové rozhraní pro správu front a dalších vlastností, které se týkají posílání zpráv [14].

2.3 Graphite

Mladý projekt, který vznikl v roce 2006, je majiteli hrdě nazýván, jako projekt připravený pro nasazení do produkčního prostředí. V roce 2008 byl vydán pod licencí Apache 2.0. Některé velké společnosti, jako je GitHub, Reddit a například Booking.com, ho úspěšně používají v produkčním prostředí a jsou s ním spokojeni. Sami vývojáři prezentují, že Graphite dělá jen 2 věci, a to je uložení dat a generování grafů na vyžádání s hodnotami v reálném čase. Je to tedy nástroj, který se nezabývá aktivním sbíráním dat. Obecně lze říci, že pasivně čeká na data, která mu zašle nějaký externí zdroj. Jeho vývoj je velmi živý, na webových stránkách se zdrojovým kódem lze vidět aktualizace staré pouze několik dní, což lze jistě považovat za velkou výhodu. V počátku byla použita pro uložení dat RRD, která nesplňovala požadavky Graphitu z několika již zmíněných důvodů. Proto používá databázi Whisper, která problémy RRD řeší. Graphite implementuje optimalizaci, která ukládá přijatá data do dočasné paměti, a až ve chvíli, kdy je dat požadované množství, je zapíše jako jednu větší dávku na perzistentní médium. Díky své historii je Graphite kompatibilní s RRD. Poskytuje rozhraní, které stačí implementovat a nahradit tak použitou databázi. Hlavní předností tohoto systému je možnost vykreslení dat z již zmíněné dočasné paměti, která z něj dělá poměrně silný nástroj. Je implementovaný pomocí jazyka Python verze 2 a jeho architektura je znázorněna na obrázku 2.2. Data přijatá systémem se v jeho terminologii nazývají metriky. Což je trojice, která obsahuje název dat, čas a samotnou hodnotu. Název metriky chápeme jako řetězec bez mezer, který je složen z několika podřetězců rozdělených tečkami. Je vytvořen jako sada několika různých démonů, kdy každý má přidělený jeden dílčí úkol. Pro použití není nutné mít spuštěny všechny démony. To dělá Graphite velmi dobře škálovatelným. Pro škálování stačí prakticky spustit v několika instancích.

Jak je z architektury zřejmé, tak Graphite se skládá ze tří základních částí, které zde postupně popíši:

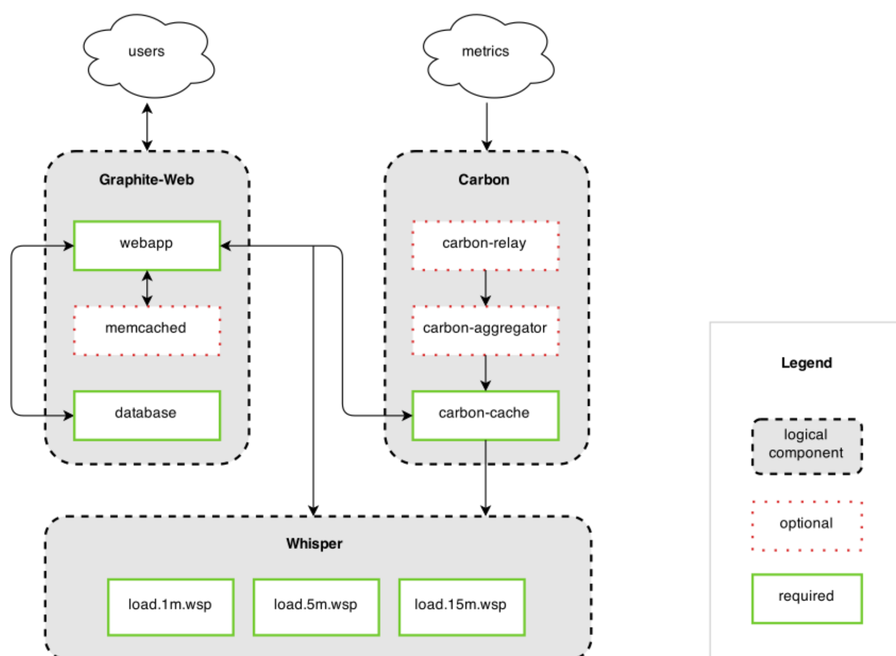
Graphite Web uživatelské rozhraní, které poskytuje veškerá přijatá data

Carbon část, která se stará o uložení a o přípravu dat pro uložení

Whisper již několikrát zmíněná databáze včetně databázových souborů 2.1.2

2.3.1 Graphite Web

Webová aplikace, která je také naprogramovaná v jazyku Python verze 2, poskytuje uživateli plný přístup ke všem datům, které systém Graphite přijal. Tato aplikace je závislá na několika knihovnách, které jsou pro Python běžné, ale jedna z nejdůležitějších je knihovna Cairo, která poskytuje generování grafů ze získaných dat. Další důležitou knihovnou je Django, což je knihovna komunikující s databází. V tomto případě relační databází, která je využita jen pro uložení provozních informací webové aplikace. Ukládá předpisy pro generování grafů, informace o uživatelích a další maličkosti. Již jsem zmínil sílu ve škálovatelnosti nástrojů systému Graphite, této vlastnosti se nevyhne ani webová aplikace, která také umožňuje škálovat tzv. do šířky a to spuštěním několika instancí s příslušnou konfigurací. Pro ověření do aplikace lze použít databázi



Obrázek 2.2: Graphite Architektura [15]

LDAP, která ve většině prostředí poskytuje uživatelské účty. Pro rychlejší odezvu při reakci na dotazy získané prostřednictvím API, ale i přes webové rozhraní lze spustit s příslušným nastavením databázi v operační paměti jménem Memcached, která slouží jako dočasná paměť pro data, která uživatel často vyžaduje.

2.3.1.1 Vykreslování grafů

Webové rozhraní pro konzumenty těchto grafů je na první pohled velmi jednoduché. Umožňuje prakticky jen sestrojení grafů a jejich zobrazení v jednoduchém přehledu. Sestrojené grafy se ukládají pro každého uživatele samostatně a každý si tak může nastavit skupinu grafů, kterou chce zobrazit a mít na jednom místě. Uživateli poskytuje možnost si vytvořit několik ploch, které obsahují jeho vybrané grafy a nastavit si jejich automatické aktualizování, což může být výhodná funkce, například pro zobrazení nejdůležitějších grafů na televizi. Pro generování grafů lze použít velkou škálu funkcí, které upravují data a připravují je pro vykreslení. Nejdůležitější funkce zde představím. Podrobnější informace jsou ve zdroji [16].

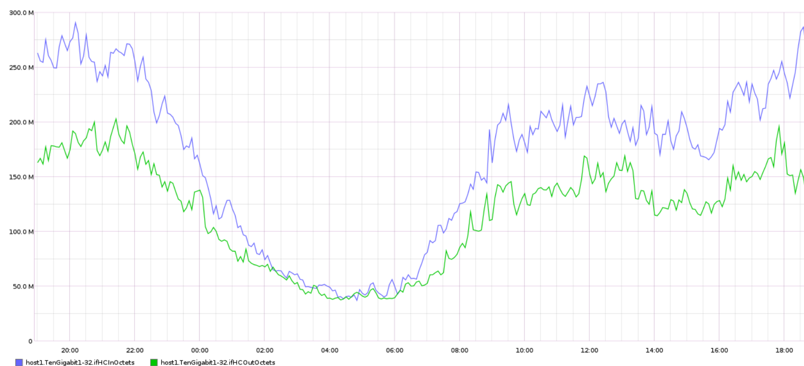
alias nastaví název zdroje dat zobrazený v legendě grafu

scaleToSeconds vrátí změnu hodnoty za sekundu (případně za více sekund podle parametru)

asPercent spočítá procentuální podíl na základě parametrů

derivative pro přepočítání hodnot z počítadel na rozdíly

keepLastValue pokračuje v kreslení řady, pokud chybí mezilehlá data



Obrázek 2.3: Graf vygenerovaný pomocí Graphite Web

```
Použité funkce pro vykreslení grafu:
alias("host1.TeGiEth0-1.ifHCInOctets").scaleToSeconds(1)
    .nonNegativeDerivative(název.metriky)
```

2.3.1.2 API

Webová aplikace poskytuje URL API, pomocí kterého lze získat data z databáze Whisper a z paměti démona „carbon-cache". API je přístupné přes URL:

```
http://GRAPHITE_HOST:GRAPHITE_PORT/render
```

Graf získáme jako odpověď na HTTP dotaz typu GET s příslušnými parametry. Parametry se předávají prostřednictvím běžných GET parametrů přímo v URL.

Vykreslení grafu je jedna z hlavních úloh API a nabízí velkou škálu možností jeho úpravy. Některé z nich zde popíši. Ostatní parametry jsou k nalezení v [17, kap. The Render URL API] a týkají se hlavně detailů při vykreslování, jako jsou barvy, velikost a popisky.

target název metriky, který může být obalen funkcemi pro úpravu získané hodnoty, které jsou popsané v 2.3.1.1, v názvu metriky můžeme použít znak hvězdička, který se expanduje na všechny podstromy názvu metriky, nebo lze použít výčet těchto hodnot

from/until jak už z názvu plyne, tak tyto parametry nám umožňují určit, v jakém časovém úseku chceme data získat; máme možnost vložit relativní časový úsek pomocí záporné hodnoty v parametru from, pak se parametr počítá z aktuálního času, anebo absolutní s použitím obou parametrů

format jako formát výstupních dat můžeme požadovat obrázek formátu PNG, SVG, PDF a nebo například data ve formátu JSON či CSV, ale není problém získat data ve formátu RAW, tedy v binární podobě

template šablona, která obsahuje informace o vykreslovaném grafu a přijímá parametry, které použije pro nalezení konkrétních zdrojů dat

2.3.2 Carbon

Jak už jsem zmínil v předchozí části, Carbon se skládá ze 3 démonů. Každý má svůj dílčí úkol, a proto není nutné vždy provozovat všechny. Samotný démon „carbon-cache" je ale nejdůležitější a stará se o ukládání hodnot na perzistentní médium. Každý z těchto démonů může být spuštěn více než jednou, čímž lze rozložit zátěž pomocí škálovatelnosti do šířky. Tyto démoni jsou naprogramováni v jazyku Python verze 2. Jsou konfigurovatelní pomocí několika konfiguračních souborů, které zmíním v následujících řádcích. Každý z těchto démonů přijímá data ve formátu, který jsem představil v úvodu kapitoly. Tato data neboli v terminologii Graphitu metriky mohou být přijaty prostřednictvím třech dostupných metod: plaintext, pickle protokol a AMQP. Metoda plaintext umožňuje zaslat data, například pomocí jednoduchého nástroje netcat, ale je omezen pouze na jednu metriku. Pickle protokol poskytuje ve většině případů větší komfort, protože umožňuje zaslání více metrik v jednom paketu a tím šetřit výpočetní zdroje. Poslední možností je AMQP, což je protokol umožňující asynchronní zasílání zpráv s daty a budu se mu věnovat později.

2.3.2.1 Démon carbon-relay

Pokud použijeme démona „carbon-realy", je vždy předsunut před ostatní části a přijímá data. Tento démon poskytuje dvě funkce týkající se škálovatelnosti a zálohování. Jedna z funkcí je replikace, která nám umožňuje zasílání metrik do více démonů „carbon-cache", kteří data uloží a dále zpracují. Druhá funkce je tzv. „sharding" neboli rozdělení příchozích dat na základě předem definovaných funkcí, které mohou být dynamické, takže založené na hašovací funkci a nebo na pravidlech, které si předem určí administrátor ve speciálním konfiguračním souboru s názvem „relay-rules.conf". V tomto souboru zapisujeme pravidla, která na základě regulárního výrazu a názvu metriky určí kam uložit příslušná data.

2.3.2.2 Démon carbon-aggregation

Tento démon se spouští mezi démony relay a cache. Jeho funkcí je připravení dat pro vypočítání agregační funkce. Ovlivňuje tak zátěž pro operace zápisu a čtení na perzistentním médiu, protože data mohou přicházet častěji než je předepsáno, jsou přeposílána k uložení až po dosažení konce předepsaného intervalu, takže se stihnou agregovat a provádí se pouze jeden zápis. V případě, že by taková data chodila přímo do „carbon-cache", tak je přepisuje stále dokola a vygeneruje tak více operací zápisu. Pro konfiguraci agregace používá konfigurační soubor „aggregation-rules.conf", který obsahuje pravidla v následujícím tvaru:

```
output_template (frequency) = method input_pattern
```

Listing 2.3: Agregační pravidlo

output_template
název cílové metriky pro její uložení

frequency interval, po kterém má být hodnota s novým názvem zaslaná pro uložení

method metoda, která se aplikuje na přijatá data, na výběr jsou average, sum, min, max a last, ve výchozím stavu se použije average

input_pattern vzor pro určení metrik agregovaných tímto pravidlem

Pomocí těchto pravidel lze například sečíst několik metrik, které se uloží jako jedna. Tuto funkci můžeme využít v situaci, kdy chceme ukládat počet dotazů přicházejících na více serverů, potom stačí do vzoru pro použití pravidla vložit na příslušné místo hvězdičku a započítají se nám všechny metriky. Další využití v síťovém prostředí je uložení maximálních hodnot během dne. Můžeme ukládat například hodinové maximální hodnoty počtu zpracovaných paketů na rozhraní. Další silnou funkcí jsou proměnné v rámci jednoho pravidla, můžeme tak ve vzoru použít klíčové slovo ve špičatých závorkách a potom použít tuto proměnnou v názvu cílové metriky.

2.3.2.3 Démon carbon-cache

Nejdůležitější ze všech démonů je „carbon-cache" starající se o samotné ukládání dat. V této funkci mu napomáhá možnost uložení dat do dočasného úložiště v operační paměti a tudíž je velmi rychlý. Díky možnosti poskytnutí hodnot z dočasné paměti pro další zpracování je tento nástroj, co se využití týče celkem výkonný a vhodný do prostředí s velkým množstvím dat. Protože tento démon je základním kamenem, musí být vždy spuštěna alespoň jedna jeho instance. Administrátoři jistě uvítají možnost nastavení maximálního počtu hodnot pro uložení do dočasné paměti, což nám napomáhá využít operační paměť a její výkonnostní potenciál bez toho, abychom ji přetížili. Další vlastností, která je pro administrátora výhodná, je možnost nastavení maximálního počtu zápisů na perzistentní médium za sekundu. Pro pokročilejší správu existuje možnost přístupu do právě spuštěné instance pomocí nástroje SSH. Tyto vlastnosti se mimo jiné konfigurují prostřednictvím souboru „carbon-cache.conf". Tento démon se také stará o vytváření databázových souborů pro data, které probíhá průběžně při přijetí prvních dat. Databázové soubory mají pevný předpis, který musí být znám už při jejich vytvoření. Tento předpis se konfiguruje v souboru „storage-schemas.conf" tak, že určíme, jak dlouho a jak přesná data chceme ukládat. Jako například patnácti minutové průměry, které chci ukládat sedm dní. Každý z těchto předpisů obsahuje regulární výraz, který se aplikuje na název přijaté metriky a dle přesnosti se použije následující specifikace (použije se nejpřesnější). Jak jsem naznačil dříve, tak je pro uložení dat s delším intervalem je nutné určit funkci pro agregaci a pro předání archivům k uložení

v rámci jednoho souboru. Tuto funkci a několik dalších parametrů konfigurujeme pomocí souboru „storage-aggregation.conf". Zde je také už známý parametr *xFileFactor*, který definuje kolik nulových hodnot může být v množině, pro kterou použije agregační funkci. Ze zmíněné vlastnosti plyne, že v této databázi je použitá agregační funkce pro celý soubor a pro všechny archivy, zatímco v RRD je použita funkce pro každý archiv.

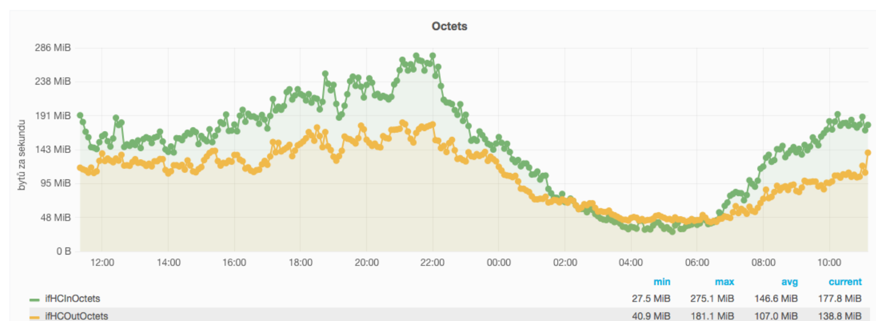
```
[all_min]
pattern = \.min$
xFilesFactor = 0.1
aggregationMethod = min
```

Listing 2.4: Agregační pravidlo v souboru storage-aggregation.conf

Metriky končící tečkou a řetězcem „min" používají agregační funkci „min" a při jejich vyhodnocení můžou obsahovat až 10 % neznámých hodnot.

2.4 Grafana

V krátkosti představím nástroj Grafana, který v současnosti patří mezi velmi oblíbené generátory grafů. Hlavně protože plně podporuje generování grafů z nástroje Graphite pomocí URL API poskytnutého prostřednictvím Graphite Web, včetně všech nabízených funkcí. Graphite není jediným podporovaným zdrojem dat, jsou jimi například i databáze InfluxDB, Elasticsearch a nebo třeba nástroj Prometheus. Grafana je rozdělena do několika částí, kdy hlavní z nich je tzv. Dashboard, což je obrazovka s vygenerovanými grafy. Dashboard udržuje pohromadě nastavení grafů a jejich množinu předkonfigurovanou uživatelem. Uživatel má k dispozici šablony, které přijímají prostřednictvím proměnných zdroje konkrétních hodnot později použitých v předpisech pro generování grafů. Uživatel tak jen určí šablonu, která může obsahovat množinu předpisů grafů, a to včetně použitých funkcí pro vykreslovaná data, a dále mění jen parametry šablony, jako například zdrojového hosta. Počet těchto obrazovek pro uživatele není omezen a lze je i exportovat a nebo importovat některé z veřejně dostupných. Jednou z výhod grafů vygenerovaných pomocí Grafany je jejich uživatelská přívětivost, a to jak z pohledu grafiky, tak z pohledu možnosti interaktivně s nimi pracovat. Protože Grafana poskytuje možnost vývoje externích pluginů, tak vznikla celá škála různě orientovaných rozšíření, které lze doinstalovat a získat tak mnoho dalších možností [18].



Obrázek 2.4: Graf vygenerovaný nástrojem Grafana

2.5 Management information base

Zkráceně MIB je databáze strukturovaně popisující hodnoty dostupné pro konfiguraci a čtení na síťovém zařízení. Databáze má stromovou strukturu a na konci každé větve, tedy v listech, se nachází hodnota. Tato databáze je původně navržena jako samostatná část, kterou může využívat více služeb. Její hodnota může mít různé datové typy, které jsou specifikovány v [19]. Každý list nebo podstrom lze jednoznačně určit pomocí identifikátoru, který se nazývá OID. Protože OID je jen řetězec čísel rozdělených tečkami, co číslo to uzel stromu, tak existuje možnost OID překládat na textové řetězce srozumitelnější pro uživatele a napomáhající mu při správě. Zařízení může implementovat různé množství těchto MIB, které poskytují nejrůznější data. MIB mohou být samozřejmě vytvořeny konkrétním výrobcem zařízení, kterému musí být přidělen vlastní podstrom, abychom předcházeli situacím, kdy více výrobců poskytne různou hodnotu za stejným OID [20]. Objekty definované v databázi MIB jsou definované pomocí jazyku ASN.1 [21].

```
1 ISO
1.3 ORG
1.3.6 Department of Defense
1.3.6.1 Internet
1.3.6.1.2 IETF Management
1.3.6.1.2.1 SNMP MIB-2
1.3.6.1.2.1.1 SNMP MIB-2 System
1.3.6.1.2.1.1.3 sysUpTime
sysUpTime OBJECT-TYPE
          SYNTAX  TimeTicks
          ACCESS  read-only
          STATUS  mandatory
          DESCRIPTION
                "The time (in hundredths of a second)
                since the network management portion
                of the system was last re-initialized."
          ::= { system 3 }
```

Listing 2.5: Tvar proměnné sysUpTime v MIB

2.6 Simple Network Management Protocol

Už název protokolu napovídá, že byl stvořen primárně pro potřeby správy síťových zařízení. SNMP značně pomohl řadě správcům síťových infrastruktur. Do té doby se v případě potřeby musela většina správců na zařízení připojit, provést změnu konfigurace a potom zařízení restartovat. S příchodem SNMP se otevřela možnost vytvoření nesčetného množství nástrojů pro správu zařízení. Ačkoliv jde o bez mála 30 let starý protokol, dokázal si vydobýt uznání a i v dnešní době je velmi využívaný. Už v roce 1990 vznikl dokument [22], který doporučuje implementaci SNMP v každém síťovém zařízení. V dnešní době bychom jen velmi těžko hledali síťové zařízení nebo aplikaci pro jejich správu, která tento protokol nepodporuje. Další nespornou výhodou je jeho volné využití a nekontrolovatelnost jedním subjektem, samozřejmě výjma standardizace.

A protože byl protokol navržen pro méně výkonné zařízení, než známe ze současnosti, tak je velmi šetřivý, co se týče zdrojů, a to je vlastnost, která se může ukázat jako velmi výhodná v nastupující době IoT.

SNMP používá jako protokol 4. vrstvy ISO/OSI modelu UDP, který nezaručuje doručení paketů, ale je to jeden z nejrychlejších způsobů komunikace. Od verze s označením 2c je možné posílat tzv. trapy ze strany serveru, tedy ve většině případů ze síťového zařízení. Trapy jsou pakety s hodnotou, které jsou dále zpracovány na straně klienta. Protokol ve výchozím nastavení používá pro komunikaci UDP port 161 a pro přijímání trapů používá UDP port 162. Protokol je asynchronní a založený na modelu klient-server. Klient pro komunikaci je velmi jednoduchý program pro odeslání dotazu směrem k síťové zařízení, na kterém je spuštěn tzv. snmp-agent, který je zpracovává. Každý SNMP paket obsahuje PDU, jehož struktura je předem definována jeho typem. Vedle PDU paket obsahuje další provozní informace lišící se dle použité verze [23]. Aktuálně se používají 3 verze z nichž, každá poskytuje několik nových možností [24]:

verze 1 původní verze protokolu, která při svém vzniku podporovala větší škálu protokolů 3. vrstvy modelu ISO/OSI, její vznik je datován do roku 1988

verze 2c umožňuje poslat dotaz na celý podstrom hodnot a možnost zasílání trapů

verze 3 přidává hlavičku do síťového datagramu, pomocí které umožňuje ověření a šifrování komunikace

Typy PDU zpráv pro komunikaci pomocí protokolu SNMP:

SET příkaz k nastavení hodnoty konkrétního OID nebo množiny OID, které odesílá strana klienta směrem k serveru, ten tyto změny provede atomicky a následně odešle klientovi seznam OID s novými hodnotami

GET žádost o hodnotu konkrétního OID směrem od klienta k serveru, ten vrátí datově stejný paket, ve kterém nastaví správnou hodnotu OID a změní jeho typ

GETNEXT žádost o hodnotu a OID, které následuje ve stromové struktuře za OID vloženém v dotazu, OID jsou seřazena numericky, pomocí této funkce je možné iterovat přes celou MIB poskytovanou agentem a ve verzi 1 to byla jediná možnost, jak se jednoduše dostat k větší množině dat za cenu relativně velké zátěže

GETBULK od verze 2c se lze dotázat na celý podstrom od konkrétního OID, tato možnost si vyžádala přidání parametru, který určuje množství hodnot zařazených do odpovědi tzv. „max-repetitions“, abychom se tedy dostali na další data podstromu, musíme zaslat další bulk dotaz na poslední OID získané v předchozí odpovědi, vznikne tak celá kaskáda

bulk dotazů; další nutností bylo přidání parametru „non-repetitions“, který určuje počet opakování bulk dotazů v případě neobdržení odpovědi [25]

RESPONSE označení všech paketů, přijatých jako odpověď na zmíněné dotazy, obsahuje parametry nesoucí informace o případné chybě

TRAP asynchronní zpráva odeslaná ze snmp-agenta informuje klienta například o rozhraní, které změnilo svůj stav, nebo o ztrátě jednoho z více zdrojů, protože je spojení iniciováno z agenta, tak je nutné na cílovém klientovi naslouchat na předem určeném portu – ve výchozím nastavení zmíněný port 162

INFORM z důvodu rychlosti protokol využívá UDP komunikaci, která bohužel nemá žádnou možnost ověřování doručení, takže vznikl tento typ PDU, díky kterému si mohou předat informaci o přijetí paketu, toto se konkrétně děje až od verze 2c a v případě PDU typu TRAP

```
Simple Network Management Protocol
    version: v2c (1)
    community: communityro
    data: get-request (0)
        get-request
            request-id: 530766591
            error-status: noError (0)
            error-index: 0
            variable-bindings: 1 item
                1.3.6.1.2.1.1.3.0: Value (Null)
                    Object Name: 1.3.6.1.2.1.1.3.0
                    Value (Null)

Simple Network Management Protocol
    version: v2c (1)
    community: communityro
    data: get-response (2)
        get-response
            request-id: 530766591
            error-status: noError (0)
            error-index: 0
            variable-bindings: 1 item
                1.3.6.1.2.1.1.3.0: 309687139
                    Object Name: 1.3.6.1.2.1.1.3.0
                    Value (Timeticks): 309687139
```

Listing 2.6: Příklad SNMPv2c dotazu a odpovědi

2.6.0.4 Verze 1

První verze protokolu SNMP byla kritizována pro svou nulovou bezpečnost a nemožnost jakéhokoliv ověření dotazujícího klienta. Jediná možnost ověření probíhala prostřednictvím řetězce znaků, který byl přes síť přenášen v čitelné a otevřené podobě, řetězec se označuje jako „community“. Cílový agent měl sice informaci o přístupu „community“, ale ta se týkala jen možnosti rozlišit mezi přístupy jen pro čtení, jen pro zápis nebo pro obě možnosti, ale téměř každý si mohl v síti odposlechnout případný řetězec s příslušnými právy a využít ho. Vzhledem ke stáří této verzi lze chápat implementaci bezpečnosti založenou na důvěře, protože v té době byl Internet pouze pro několik vyvolených, kteří ho používali ve svém zájmu a tudíž neměli ambice mu jakkoliv škodit. Zabezpečení lze tedy dosáhnout jen prostředky, které nepřísluší protokolu SNMP [26].

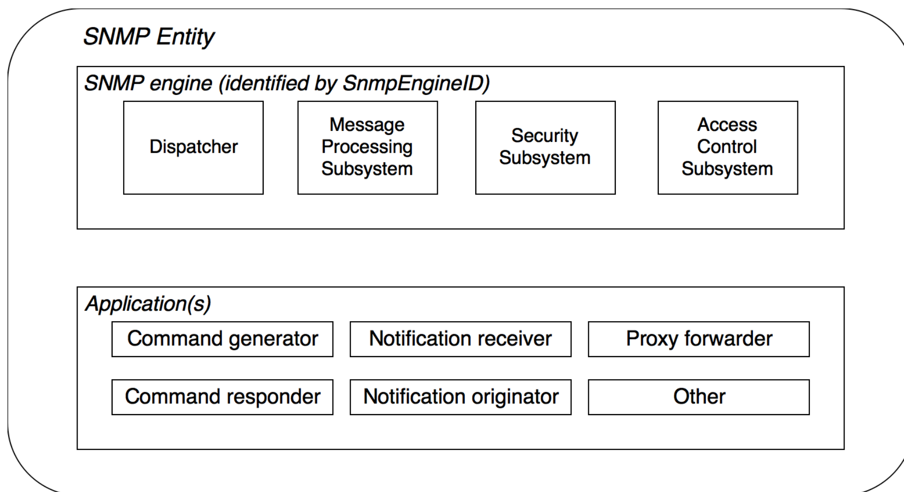
2.6.0.5 Verze 2c

Tato verze je označována s přidaným „c“, které znamená ověření pomocí „community“, jako tomu bylo u verze 1, takže ohledně bezpečnosti nepřináší žádné výhody. Je specifikována několika dokumenty. Verze označovaná pouze jako verze 2 přinesla komplexní řešení bezpečnosti a zásadní rozdíly ve strukturách paketů, což jsou pravděpodobně hlavní důvody, proč se tato verze nikdy nestala zajímavou pro veřejnost. Protokol verze 2 byl definován pomocí více než 400 stránkového dokumentu, což je oproti 36 stránkám dokumentu popisující verzi 1 značný rozdíl [27].

2.6.1 Verze 3

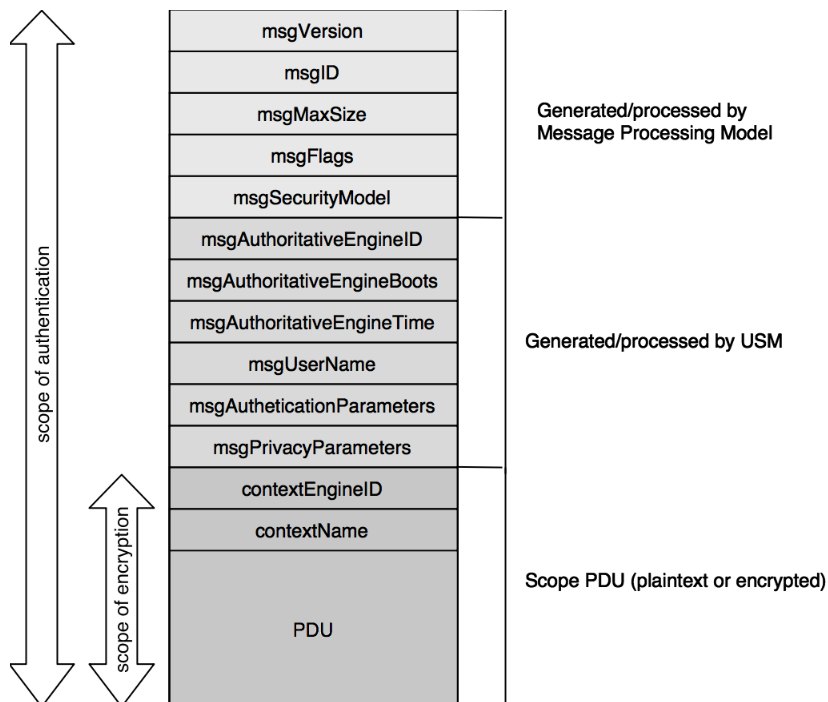
Jak už jsem několikrát zmínil, tak bezpečnost a obecná zranitelnost byla hlavním problémem předešlých verzí protokolu SNMP. Verze 3 se tedy zaměřuje hlavně na bezpečnost a neposkytuje tedy žádný nový typ PDU, ale plně podporuje všechny operace, které poskytují předešlé verze. V některých částí jen lépe interpretuje existující vlastnosti, jako je například změna terminologie. V předešlých verzích bylo zvykem nazývat protilehlé strany „manager“ a „agent“, ale verze 3 přináší jeden název pro obě strany a to je „entita“. Každá entita obsahuje jednu nebo více SNMP aplikací a SNMP „engine“. Tato architektura umožňuje rozdělení částí SNMP systému, které umožňují implementaci zabezpečovacích mechanizmů [28]. Tento fakt zapříčinil pomalé zavedení této verze do běžných zařízení, protože z pohledu uživatele se nic zásadního nezměnilo (stále provedu jeden z dotazů a dostanu odpověď), ale z pohledu programátorů, kteří implementují tuto verzi, nastaly dost radikální změny.

SNMP engine má za úkol přijímat a odesílat zprávy, připravit data pro odeslání, získat data z již přijatých zpráv, poskytovat ověření, šifrování a jejich dešifrování. Pomocí přístupových pravidel lze ověřit přístup k databázi MIB. Díky této modularitě je jednoduché implementovat podporu předešlých verzí, protože stačí zaměnit část, která zpracovává přijatá a odesílaná data. SNMP aplikace poskytuje tedy zbývající vlastnosti, jako jsou generování dotazů a odpovědí, generování trapů a přijímání trapů. Dále může poskytovat například proxy pro přeposílání.



Obrázek 2.5: Ukázka SNMPv3 entity (překresleno podle předlohy v [26, str. 76])

Pro verzi 3 engine poskytuje bezpečnostní model, který je založený na uživateli (USM). Tento model zařizuje ověření uživatele a šifrování dat. Každá entita udržuje tabulku, která obsahuje uživatele, jejich práva a klíče pro komunikaci s nimi.



Obrázek 2.6: Ukázka formátu zprávy SNMPv3 (překresleno podle předlohy v [26, str. 79])

Zpráva obsahuje několik konfiguračních polí, z níž ty nejdůležitější popíši a zbývající lze dohledat v [26, kap. 3].

msgSecurityModel
pole popisující bezpečností model, pro který je tato zpráva určena, v dnešní době je k dispozici hodnota 1, 2 a 3 pro verze SNMP

msgUserName uživatel pro ověření a autorizaci

msgAuthenticationParameters
obsahuje NULL, pokud není použitý žádný typ ověření a nebo HMAC, momentálně RFC popisuje použití hašovacích funkcí MD5 nebo SHA, klíč pro vypočítání HMAC musí znát obě strany komunikace

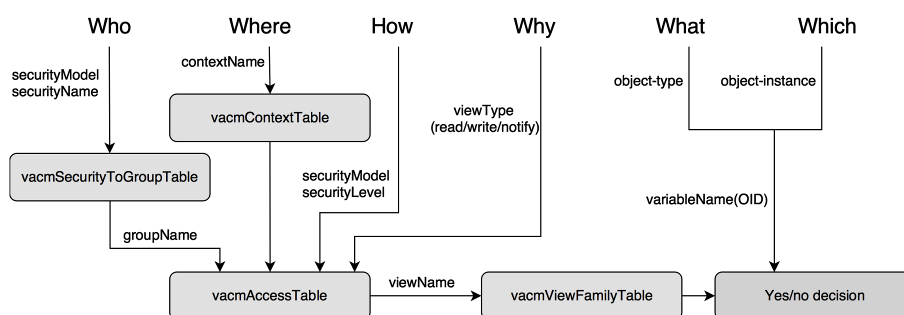
msgPrivacyParameters
obsahuje NULL, pokud není použité žádné šifrování nebo obsahuje inicializační informaci pro blokovou šifru CBC-DES, klíč pro šifrování musí znát obě strany komunikace

contextEngineId
jedinečný identifikátor odesílatele

contextName jméno kontextu, které je použité systémem VACM

scopedPDU část obsahující kromě provozních dat ještě samotné PDU, které známe z předešlých verzí

Modul poskytující řízení přístupu uživatelů k MIB se nazývá View Access Control Model. Udržuje 4 tabulky, které obsahují informace o tom, jací uživatelé a za jakého předpokladu, jsou oprávněni používat určité části MIB. Informace určují práva přístupu, které jsou definované na základě parametrů komunikace.



Obrázek 2.7: VACM vyhodnocení přístupu (překresleno podle předlohy v [26, str. 83])

Použité tabulky pro vyhodnocení přístupu [29]:

vacmContextTable
: kolekce spravovaných objektů obsahující přístupová omezení

vacmSecurityToGroupTable
: pomocí securityModule a securityName poskytne jméno příslušné skupiny

vacmAccessTable
: nejdůležitější tabulka systému obsahující informace o tom, ke kterému daný contextName, groupName s použitým parametrem securityModel a securityLevel mohou přistupovat viewName

vacmViewFamilyTable
: tabulka umožňující překlad viewName na příslušný podstorm v MIB

2.6.2 Sada nástrojů Net-SNMP

Pokud se administrátor někdy zabýval protokolem SNMP, určitě se setkal s touto sadou nástrojů. Je vydávána pod licencí BSD a je dostupná pro všechny běžné operační systémy. Tato sada obsahuje několik nástrojů pro protokol SNMP, jako jsou jednoduché aplikace pro všechny typy dotazů s podporou všech verzí SNMP a stejně tak několik modulů pro další jazyky. Hlavním přínosem je ale dlouho vyvíjená a jedna z nejpopulárnějších knihoven pro jazyk C potažmo jazyk C++ pracující s tímto protokolem. Má plnou podporu protokolu IPv6, ale i IPX a dalších. Protože s protokolem SNMP úzce souvisí již zmíněná databáze MIB, tak přináší i jednoduché aplikace, které poskytují překlady a další práci s touto databází. Tuto knihovnu používají i knihovny určené pro další jazyky, jako je například Java [30].

2.7 Zabbix

Zadání této diplomové práce by splnil dohledový systém Zabbix, který je vyvíjen bezmála 10 let a je to relativně úspěšný projekt. Poskytuje komplexní řešení dohledu a generování grafů z dat získaných několika různými způsoby. Díky jeho dlouhé existenci lze předpokládat, že architektura, kterou vývojáři zvolili, byla v mnoha ohledech velmi výhodná. Tuto architekturu se pokusím více přiblížit, protože byla jednou z hlavních inspirací při návrhu dohledové části. Avšak dohledový systém Zabbix má několik nedostatků, které se v této práci pokusím řešit lépe.

2.7.1 Získávání dat

Zabbix je komplexní řešení dohledového systému pro malou až středně velkou síťovou infrastrukturu. Tento dohledový systém poskytuje několik úrovní dohledu. Jedna z nejdůležitějších úrovní takového systému je sbírání dat, které lze v Zabbixu provozovat několika různými způsoby, data je možné sbírat aktivně nebo pasivně (čekat na jejich přijetí). Samozřejmá je také podpora získání dat

přes SNMP, které je cílem této práce. Zabbix od nedávno zveřejněné verze podporuje dotazování pomocí tzv. bulk dotazů, které šetří čas, výkon cílového zařízení a šířku síťového pásma. V terminologii Zabbixu se hodnota, která se pravidelně sbírá, nazývá „item“ [31, kap. Zabbix concept]. Nejdůležitější typy itemů:

Jednoduché kontroly
ICMP, TCP SYN, DNS překlad a několik dalších

SNMP běžné GET dotazy, bulk dotazy s podporou rodičovského OID

Zabbix agent program spuštěn na serveru, se kterým přímo komunikuje Zabbix server a poskytuje hlubší vhled do OS, který je předmětem dohledu

2.7.1.1 Ukládání dat

Zabbix využívá jako úložiště jednu z relačních databází, jako je PostgreSQL, Oracle nebo MySQL. Tyto databáze jsou vhodné pro ukládání strukturovaných dat, které se pokud možno příliš často nemění. Je sice známo, že i tyto databáze jsou schopny dosáhnout relativně velkého výkonu, ale už z povahy věci si lze představit efektivnější způsob ukládání těchto časově závislých dat pro generování grafů. Zabbix tato data ukládá do tabulek dle datového typu, takže například všechna data typu kladné číslo jsou v jedné tabulce, která obsahuje vazbu na příslušný item, časy příchodů a samotnou hodnotu. Tato data je nutné průběžně odstraňovat prakticky každý den, což je ve velkém množství relativně náročné pro tento typ databází. Při odstraňování dat v relačních databázích vznikají ve většině případů prázdné mezery, které je nutné redukovat, případně recyklovat, což je problém jednotlivých databází a není předmětem této práce.

2.7.2 Vyhodnocení dat

Dalším hlavním pilířem každého dohledového systému je vyhodnocení již získaných dat, tato část je v Zabbixu zcela oddělena od části, která data obstarává. Vyhodnocovací část obsahuje výraz, který má několik možných stavů a to jsou splněný, nesplněný, potažmo neznámý. Tento výraz je součástí entity, která se v terminologii Zabbixu nazývá „trigger“, což není nic netypického, triggery známe i z dalších míst v oblasti informačních technologií. Výraz, který je vyhodnocován při kontrole stavu příslušného triggeru, má následující syntaxi:

```
{název.itemu.funkce()} operátor {název.itemu.funkce()}
```

Řetězec název.itemu se odkazuje na jednoznačně identifikovatelný item. Operátor dle očekávání můžeme nahradit za jakýkoliv logický nebo matematický operátor a funkce je jedna z nabízených funkcí, která poskytuje předzpracování dat. Může být například last, která vrátí poslední získanou hodnotu itemu. Výrazy lze samozřejmě libovolně kombinovat a větvit do složitějších výrazů, což se projeví poklesem výkonu. Při změně stavu triggeru se v Zabbixu vygeneruje tzv. „event“, což je událost, která se zpracuje v jiné části.

2.7.3 Vyhodnocení událostí

V této části Zabbixu se zpracovávají již zmíněné události, které jsou generovány několika způsoby. Pro nás je nejdůležitější způsob změna stavu triggeru. Ale Zabbix podporuje i funkci, jako je „discovery", tedy nalezení hostů dle zadaných kriterií, například odpovídající IP adresy v zadaném rozsahu a podobně. Na základě těchto událostí se vygeneruje příslušná notifikace, log nebo akce typu spustit příkaz.

2.7.4 Generování grafů

Dohledový systém Zabbix podporuje generování grafů z dat, která ukládá pro jednotlivé itemy. Tato data se využívají pro monitoring i pro generování grafů, což je jedna z jeho předností. Bohužel generování grafů, a to hlavně s historickými daty řádově několik týdnů zpět v čase, je velmi náročné na objem databáze. S představou, že se data sbírají každých 300 sekund a my chceme uchovávat několik měsíců stará data pro grafy stovkám až tisícům síťových zařízení, brzy narazíme na výkonnostní problémy, jejichž jediným řešením je ve většině případů rozdělení Zabbixu na více nezávislých uzlů.

Kapitola 3

Návrh

Kapitolu Návrh rozdělím do částí, které se týkají jednotlivých komponent celého systému. Po nastudování dokumentace nástrojů sepíši přehled jejich spolupráce a komunikace. Nejprve začnu samotným návrhem komponenty Kolektor, která data sbírá a je tedy dle mého názoru nejdůležitější částí, dále budou následovat informace o systému ukládání dat a o jejich vyhodnocování. Části systému, které bylo nutné naprogramovat od počátku, rozepíši podrobněji pro lepší představu o jejich struktuře. Na základě rozdělení do několika komponent popíši i návrh komunikace mezi ostatními systémy. Pro naprogramované komponenty bylo nutné navrhnout několik databází, které dále představím. Komponenty, které nebylo nutné programovat, popíši jen z pohledu účastníka celého systému a jejich bližší konfiguraci popíši v kapitole Realizace.

3.1 Kolektor

Rozhodl jsem se tuto část programovat v jazyku C++ potažmo C, protože je očekáván velký výkon. Bylo nutné dodržet možnost škálování pomocí paralelního sběru dat. Pro další ušetření výkonu Kolektor využívá tzv. bulk dotazů, které jsou dostupné od SNMP verze 2c, ale v případě potřeby vyšší bezpečnosti je možné využít i verzi 3. Protože zmíněné bulk dotazy vrátí celý podstrom dat, bylo nutné vyřešit otázku využití všech i přesto, že uživatel konfiguruje sběr každé hodnoty jednotlivě. Protože v případě celého podstromu není možné jednoduše zjistit, ke které komponentě se hodnota vztahuje, tak je nutné informaci nejprve získat prostřednictvím rodičovského OID a příslušného názvu komponenty. Přiblížím toto chování na příkladě, kdy chceme sbírat data o síťovém rozhraní, konkrétně počet přenesených síťových paketů.

Protože z pohledu administrátora známe pouze název rozhraní a neznáme konkrétní OID jeho dat, musíme pro získání hodnot provést vlastně dva dotazy. První dotaz získá seznam všech rozhraní a to v podobě názvů, konkrétně například „GigabitEthernet0/1“, jak je vidět na obrázku 2.6, tak zařízení v odpovědi vrátí OID s konkrétní hodnotou a není tomu jinak u dotazů typu bulk, čímž získáme tvar konkrétního OID pro každé rozhraní. V rámci jednoho podstromu existuje část OID, která se nazývá index, to je jednoznačný identifikátor této hodnoty napříč dalšími podstromy. Pokud se dále zeptáme na OID, které nám vrátí počty zpracovaných paketů, tak budeme hledat OID končící stejným indexem a tím máme zaručeno, že se získaná hodnota týká totožného rozhraní.

V tomto případě je v mé terminologii rodičovské OID to, které vrátí názvy rozhraní. Pro zobecnění přistupuji k rozhraní jako ke komponentně a k jeho názvu potom jako k názvu komponenty. Tento případ lze přenést na další zařízení či komponenty, jako jsou například teploměry zařízení pro kontrolu prostředí a podobně.

Aplikace je připravena pro spuštění na serveru jako démon, který neustále sbírá data a dále je zpracovává. Pro dosáhnutí, co možná nejrychlejšího sběru všech dat (chceme data v konkrétním čase), je potřeba data sebrat a nezabývat se jejich odesláním ve stejnou chvíli, což si můžeme dovolit za předpokladu, že k datům přidáme časovou značku jejich sebrání. Proto je Kolektor rozdělen do částí, které se starají jednotlivé problémy. Dá se říci, že ve chvíli, kdy nebude vyhovovat zasílání dat pro další zpracování pomocí AMQP, je možné ho jednoduchým krokem předělat, například pro ukládání dat do RRD nebo pro další zpracování, aniž bychom ovlivnili část zabývající se sběrem dat. V úvodní verzi musí uživatel nastavit globální interval pro sběr dat, to znamená, že není možné některá data sbírat častěji nebo naopak méně častěji. Toto omezení je nutné z důvodu dalších částí systému, například kvůli konfiguraci ukládání dat v systému Graphite, ale neměl by být problém toto chování v rámci budoucího vývoje změnit.

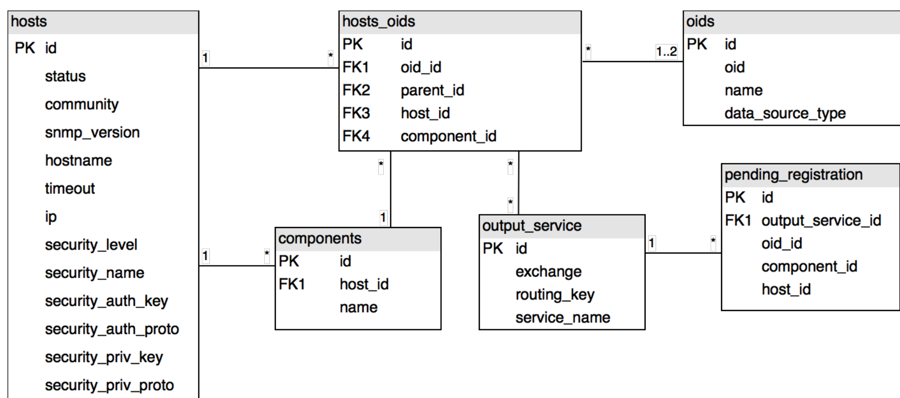
3.1.1 Konfigurace kolektoru

Protože je démon naprogramován pro jeden konkrétní úkol, tak nenabízí možnost konfigurace dat, která je bude sbírat. Tuto konfiguraci získává z relační databáze, konkrétně implementace PostgreSQL. Konfiguraci množiny zařízení určených pro sběr získává při každé iteraci sběru, což znamená, že stačí daná data přidat v určitém tvaru do databáze a Kolektor je začne sbírat automaticky. Abychom byli schopni ho konfigurovat, musel jsem naprogramovat aplikaci, která upravuje data v databázi. Aplikace přijímá strojová data pomocí JMS ve formátu JSON. Je tedy připravená pro implementaci rozhraní přívětivějšího pro uživatele, která však není součástí této diplomové práce. Tvar JMS pro konfiguraci sbírání hosta, jeho komponent a OID je zobrazen v příloze B.1.

Nedílnou součástí aplikace tohoto typu je možnost konfigurace základních parametrů pomocí konfiguračního souboru. V tomto souboru konfigurujeme především parametry připojení do databáze, parametry připojení do služby pro odesílání dat a počet paralelních vláken. Dále je zde parametr, který určuje interval mezi sběrem dat. Podrobnosti jsou popsány v kapitole Realizace. Aplikace prostřednictvím parametru přijímá cestu ke konfiguračnímu souboru.

3.1.2 Databáze

Navrhl jsem databázi, která se snaží vyjít vstříc kolektoru a umožňuje pokud možno nejednoduší přístup k datům. Hlavně protože je nutné data procházet a aktualizovat v operační paměti Kolektoru při každé iteraci. Databáze je relační. Popisuje především množiny OID, které chceme sbírat pro každého hosta. Samozřejmě nechybí konfigurační parametry hostů, kteří nám poskytují data, a to včetně timeoutu, který je následně použit pro dotazy, protože někteří hosté odpovídají pomalu, zvláště pak na větší bulk dotazy.



Obrázek 3.1: Databáze Kolektoru

hosts obsahuje nejdůležitější informace o hostovi, jako jsou ip adresa, použitá verze SNMP a další nastavení SNMP; nejdůležitějším parametrem je status, který nabývá následujících hodnot:

- **FOR_COLLECTING** status hosta, který se při další iteraci Kolektoru zařadí do množiny sbíraných dat – uloží se do operační paměti
- **COLLECTING** host, který je aktivně sbírán a je v operační paměti Kolektoru
- **FOR_DELETE** host, který je určen pro smazání a dosud je v operační paměti Kolektoru
- **DELETED** host, který je smazán z operační paměti Kolektoru a nebude se nadále sbírat
- **UPDATED** aktualizovaný host, kterého je nutné odstranit z operační paměti a znovu vyzvednout z databáze

components obsahuje hodnotu v podobě řetězce, která určuje data pro použití v případě bulk dotazů

hosts_oids udržuje vazbu mezi hostem, OID, případně rodičovským OID s komponentou

oids tabulka udržující informace o konkrétním OID, názvu a datového typu

output_services
nastavení parametrů pro odeslání dat prostřednictvím AMQP

pending_registration
čekající popis hodnot, které je nutné odesílat s příslušnými parametry

3.1.3 Komunikace

Komunikací v této části mám na mysli, jak aplikace nakládá s daty, která posbírala protokolem SNMP. Jak je zmíněno v zadání práce, tak komunikace probíhá pomocí AMQP, což je standardizovaný a kvalitně popsaný způsob sdílení dat mezi více aplikacemi. Uživatel musí mít možnost určit cílovou službu pro každá data zvlášť, protože zdaleka ne všechna data jsou vhodná pro kontrolu hodnoty a nebo naopak pro vygenerování grafu. Tuto možnost má uživatel prostřednictvím aplikace poskytované spolu s Kolektorem, která přijímá nastavení prostřednictvím JMS. Tato aplikace naslouchá na příslušném komunikačním kanále a čeká na konfiguraci, podle které upraví obsah relační databáze Kolektoru. Takže ho není nutné restartovat nebo mu jinak předávat informaci o změnách. Změnu zjistí prostřednictvím stavu hosta, který určuje, že byl upraven.

3.2 Ukládání dat a generování grafů

Pro ukládání dat jsem se rozhodl využít nástroj Graphite, který je zmíněný v kapitole Analýza. Tento nástroj využívá pro příjem dat AMQP a díky tomu může být spuštěn na jiném serveru a může se tak zvlášť škálovat, ať už na úrovni ukládání dat, nebo na úrovni příjmu dat. Kromě škálovatelnosti Graphite poskytuje výkonné ukládání dat a využití maximálního potenciálu hardwaru. Protože ukládání dat se neobejde bez vytváření souborů, které tvoří databázi, navrhl jsem tvar názvu jednotlivých metrik, které Graphite přijímá. Názvy bylo nutné určit tak, aby je uživatel při pozdějším použití dokázal nejlépe strojově identifikovat. Názvy metrik jsou složeny z řetězců bez mezer, které jsou rozděleny tečkami, tečky představují při vytvoření adresářové struktury adresář. Názvy jsou tedy složeny z následujících částí. První část je identifikátor hosta, který je v databázi Kolektoru. Nejprve připadal samozřejmě v úvahu název zařízení neboli jeho hostname, ale protože hostname si může zařízení ponechat při jeho fyzické výměně, tak jsem se rozhodl použít jedinečný číselný identifikátor. Dále pak následuje identifikátor komponenty v případě sběru pomocí rodičovského OID nebo je na druhém místě identifikátor OID, který je v případě sběru s rodičovskou hodnotou na místě třetím. Všechny identifikátory přichází prostřednictvím JMS, takže je případná nadřazená aplikace musí generovat a udržovat pro tyto účely. Pro Graphite a další části je název důležitý i z pohledu konfigurace, protože se podle něho určuje konfigurace databáze a další vlastnosti, jako je agregace či rozložení dat do více instancí. Můžeme například OID, které určuje počet paketů zpracovaných rozhraním uložit dle předepsaného předpisu a zvládneme to jedním pravidlem v konfiguraci Graphitu.

3.3 Vyhodnocování dat

Vyhodnocení získaných dat probíhá v samostatné komponentě, která data získává prostřednictvím AMQP stejně jako Graphite. Z důvodu přehlednosti a jednoduchosti jsou ve stejném tvaru, a tak máme v této části informaci pro identifikaci hosta, komponenty a i zdrojového OID. Nemusíme tedy kromě informací, které potřebujeme pro vyhodnocení, řešit žádná další data. Navrhl

jsem tuto aplikaci s inspirací systému Zabbix, který má dle mého názoru přehlednou architekturu pro vyhodnocování dat. Inspirace se týká hlavně obecného rozložení částí, které popisují samotné vyhodnocování. V této aplikaci existují dva důležité typy pracujících vláken, jejíchž počet je konfigurovatelný. Jedna sada vláken přijímá data a ukládá je do databáze v operační paměti. Druhá sada data používá pro další vyhodnocení. Zmíněná inspirace Zabbixem se projevuje hlavně tím, že jsem navrhl entity, které udržují historická data a jsou inspirovány entitami item. A protože mít itemy bez triggerů by nedávalo smysl, inspiroval jsem se i zde a vytvořil entity trigger. První sada vláken pracuje jen s itemy, ke kterým ukládá nová data, a druhá sada vyhodnocuje stavy triggerů. Samotný přístup ke zpracování itemů a triggerů je ale odlišný od systému Zabbix, už jen protože bylo nutné použít databázi, která data udržuje v operační paměti.

Protože část, která vyhodnocuje data generuje zprávy pro koncové uživatele, navrhl jsem na rozdíl od systému Zabbix možnost, která určuje konkrétní text zprávy přímo u každého triggeru. Systém Zabbix udržuje zprávy pro množinu triggerů, což je relativně velká nevýhoda. U systému Zabbix je především problém návrhu zpráv, které informují uživatele o opravení nějakého problémového stavu, protože tyto zprávy jsou obecné a díky tomu často relativně nečitelné.

3.3.1 Konfigurace

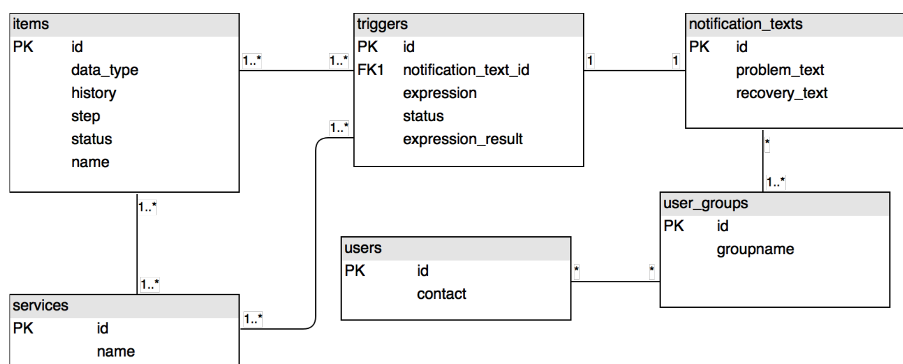
Jako konfigurační rozhraní této komponenty jsem zvolil příjem JMS zpráv, které se přímo nabízí a poskytnou tak možnost připojení komponenty do informačního systému, a tak otevírají dveře budoucímu vývoji, například uživatelskému rozhraní. Aby byla konfigurace, co možná nejpohodlnější, tak vychází z již zmíněné inspirace Zabbixem a dodržuje rozdělení entit na itemy a triggery. Itemy a triggery jsou součástí entity s názvem Dohledová služba. Dohledové služby mohou itemy a triggery sdílet, takže v případě, že budou dohledovat stejný problém, existuje pouze jeden trigger a příslušné itemy. Konfigurace, která určuje, co a jak bude vyhodnoceno, se skládá jen z množiny triggerů. Abychom věděli, jaká data bude třeba uchovávat, je nutné z této množiny získat informaci o použitých itemech a na tomto základě si je vytvořit v databázi. Pro vyhodnocení triggerů jsem navrhl několik stavů, kterých mohou nabývat. Jejich možné stavy zmíním v části o databázi.

Další konfigurace této komponenty se týká hlavně komunikace s databází, logování a samozřejmě množstvím paralelně pracujících vláken. Poskytuje také možnost nastavení komunikačních kanálů.

3.3.2 Databáze

Databáze vychází ze struktury triggerů a itemů. Je snaha k této databázi přistupovat prostřednictvím vlákna přímo určeného k udržení konzistence dat v operační paměti. Uchovává především výrazy, pomocí kterých se data vyhodnocují. Rozhodl jsem se inspirovat tvarem výrazů v systému Zabbix. V těchto výrazech jsou proměnné, které obsahují informaci pro identifikaci hodnoty, která se na příslušné místo expanduje, a funkci pro jejich předzpracování. Tato proměnná má tvar *${název.itemu.funkce(parametr funkce)}*, název itemu je zřejmý a popsaný v části o ukládání dat. Funkce je prozatím dostupná jedna, a to je

funkce *last*, která dle parametru vloží na místo proměnné historickou hodnotu. Parametr nula určuje aktuální hodnotu a dále postupujeme do historie zvětšováním hodnoty parametru. Mezi těmito proměnnými lze používat běžné logické operátory a samozřejmě i matematické operace. Přesný soubor těchto možností je popsán v kapitole Realizace. Pro ukládání dat do operační paměti bylo nutné určit, kolik historických hodnot je nutné udržovat, což je informace z perzistentní databáze. Do operační paměti se tato informace dostane pomocí externího vlákna, které aktualizuje tyto hodnoty v operační paměti a udržuje tak konzistentní stav oproti databázi relační.



Obrázek 3.2: Databáze komponenty vyhodnocující data

items obsahuje itemy s informací o intervalu sběru, o počtu potřebných historických dat a o stavu itemu

triggers triggery, které obsahují hlavně výraz pro vyhodnocení

services služba, která udržuje množiny itemů a triggerů pro snadnější správu

notification_texts texty notifikací, které jsou zasílány uživateli v případě pozitivním i negativním vyhodnocením výrazu triggeru

user_groups skupina příjemců notifikací

users uživatelé, kteří jsou použiti při posílání JMS notifikace

3.3.3 Komunikace

Již jsem zmínil, že komunikace konfigurující dohledové služby probíhá prostřednictvím zpráv JMS, které mají předepsaný tvar a pro přijetí dat itemů se používá AMQP. Prostřednictvím JMS musí být přijmut kompletní tvar dohledové služby, která obsahuje triggery, ale i zprávy, které se v případě změny stavu triggerů vygenerují a odešlou viz příloha B.4. Notifikační zprávy se odesílají také prostřednictvím JMS, které je nutné dále zpracovat externí aplikací. Tato zpráva obsahuje text přiřazený k danému triggeru a seznam cílových uživatelů. Pro uživatelskou pohodlnost jsem navrhl i spolupráci s komponentou

Kolektor. Pokud komponenta, která data vyhodnocuje, získá službu s triggery, které obsahují neexistující itemy, tak informuje Kolektor prostřednictvím speciální JMS zprávy viz B.2. Kolektor informace o itemech pošle v předepsaném tvaru zobrazeném v B.3 a zároveň zaregistruje komponentu jako příjemce dat pro tyto itemy. Pokud je Kolektor nezná, tak zařadí tento požadavek do čekající fronty (tabulka pending_registration) a zašle je až ve chvíli, kdy informace získá. Pokud neobdrží doplňující informace od Kolektoru, tak nejsou triggery těchto itemů zařazeny pro vyhodnocování. Díky této možnosti je minimalizováno množství informací, které je nutné zaslat při vytvoření dohledové služby. Komponenta si počet historických dat a interval jejich přijetí určí z výrazu v triggeru a od Kolektoru.

Kapitola 4

Realizace

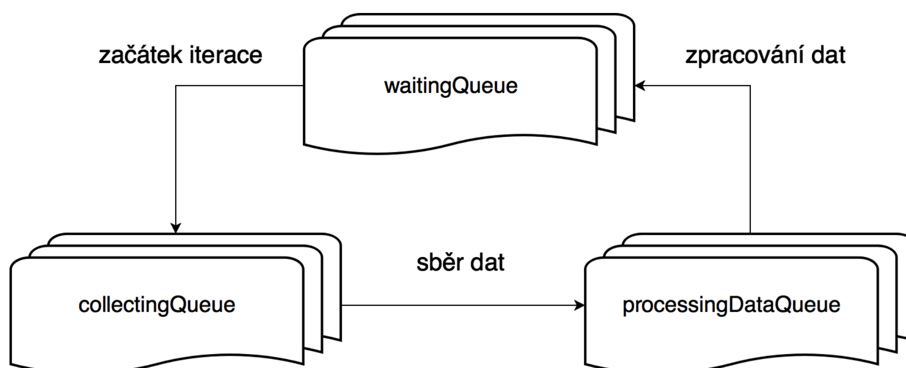
4.1 Kolektor

4.1.1 Architektura

Interní architektura Kolektoru odráží jeho funkční možnosti. Je rozdělena na dvě hlavní části, kdy jedna část data sbírá a druhá část data zpracovává. Každou z těchto činností zpracovává jiná množina vláken. Celá architektura se soustředí na práci s třemi frontami, které vlákna využívají pro předávání práce. Práce v tomto případě znamená přímo samotného hosta, který obsahuje informace o OID, které je nutné sbírat a o svém nastavení. Následující struktura je předána každému vláknu při spuštění.

```
struct DataForThread {
    ~DataForThread();
    queue<Host *> waitingQueue;
    queue<Host *> collectingQueue;
    queue<Host *> processingDataQueue;
};
```

Listing 4.1: Sdílená data vláken



Obrázek 4.1: Architektura Kolektoru

Na předchozím obrázku je zobrazena celá architektura Kolektoru. Při spuštění jsou hosté, kteří jsou určeni ke sběru, získáni z databáze a vloženi do fronty *waitingQueue*. Jak z názvu plyne, tak v této frontě čekají na dobu určenou ke sběru. Hlavní vlákno tuto dobu určí pomocí intervalu, který je nastaven v konfiguračním souboru pro celý běh instance. Následující čas sběru se vypočítá pomocí formule v následujícím tvaru:

$$(čas - čas\%interval) + interval$$

Čas je proměnná s aktuálním časem ve formátu unixtime, což vyjadřuje počet sekund uplynulých od 1. 1. 1970. Z toho plyne, že interval je počet sekund mezi každým sběrem dat. Pomocí této formule se určí čas následující iterace, na který se později v nekonečném cyklu čeká. Po dovršení této doby hlavní vlákno přesune hosty do fronty ke sběru tzv. *collectingQueue*. Z ukázky sdílené struktury mezi vlákny je zřejmé, že jsem použil implementaci fronty, která je dostupná ve standardní knihovně a pro toto použití je naprosto dostačující. Zmínil jsem, že strukturu sdílí všechna vlákna, takže je nutné postarat se o synchronizaci přístupů do jednotlivých front. Synchronizaci implementuji pomocí třech výlučných zámků tzv. *mutexů*. Přístup do každé fronty zamyká jeden. Protože je nutné je používat paralelně, jsou deklarovány globálně. Stejně tak je nutné ke zpracování hostů ve frontě umožnit přístup pouze určitému počtu vláken, pro které je dostatek práce. Tuto vlastnost jsem implementoval pomocí semaforů, což jsou ve své podstatě atomicky inkrementovatelná a dekrementovatelná čísla, která v případě nuly pozdrží vlákna a čekají na inkrementaci.

V případě dovršení vypočítaného času a přenesení hostů do fronty, ve které jsou připraveni ke sbírání, přichází na řadu první důležitá sada vláken. Počet těchto vláken je možné konfigurovat prostřednictvím konfiguračního souboru. Každé z těchto vláken po překonání semaforu a mutexu získá ke zpracování jednoho hosta. U hosta dle předepsané struktury a množiny OID, která je představena v následujících podkapitolách, získá data a uloží je k němu. Tato data jsou získaná synchronními dotazy, které poskytuje použitá knihovna ze sady Net-SNMP. Po získání všech dat hosta ho předá do fronty, ve které host čeká na další sadu vláken, která data zpracovává. Po předání hosta ke zpracování si vlákno vyžádá dalšího hosta určeného ke sběru a aplikuje stejný postup.

Druhá množina vláken je určena pouze pro zpracování dat. V konkrétním případě se zpracováním rozumí odeslání dat do určeného cíle. Tento cíl je určen názvem výstupní služby, která je získaná z databáze. Aby vlákna nemusela tuto informaci získávat při každém běhu, mají předem uloženou množinu všech výstupních služeb a použijí tedy jen tu, která má odpovídající název. Před odesláním dat do příslušné služby se musí data nejprve připravit. Data jsou serializovaná od všech OID do jedné množiny, vznikne tak množina dat k jednomu hostovi. Struktura této množiny je popsaná v kapitole o nástroji Graphite, který data přijímá. Po odeslání dat předá hosta do fronty na čekání na další iteraci.

4.1.2 Konfigurace

Konfigurace Kolektoru probíhá pomocí běžného konfiguračního souboru. Prostřednictvím tohoto souboru konfigurujeme hlavně komunikaci s databází, zmíněný interval, úroveň logování a parametry komunikace s prostředníkem pro posílání dat. Následuje výčet direktiv z konfiguračního souboru:

dbname název databáze, kterou Kolektor použije; výchozí hodnota je collector

dbport číslo portu, na kterém databáze naslouchá; výchozí hodnota je 5432

dbhost ip adresa serveru s databází; výchozí hodnota je 127.0.0.1

dbuser uživatel pro přístup do databáze; výchozí hodnota je collector

dbpassword heslo pro přístup do databáze, které musí být nastaveno uživatelem

step interval sběru; výchozí hodnota 300s

collecting_threads počet vláken, která sbírají data; výchozí hodnota 1

processing_threads počet vláken zpracovávající data; výchozí hodnota 1

rabbitmq_hostname ip adresa prostředníka pro příjem dat

rabbitmq_port port, na kterém naslouchá prostředník pro příjem dat

rabbitmq_user uživatel, kterým se ověřuje Kolektor do prostředníka

rabbitmq_password heslo předešlého uživatele

rabbitmq_content_type datový typ zpráv

log_leve úroveň logování, která může nabývat hodnot 0-7, kdy 7 je pro DEBUG; výchozí hodnota je 3 ERROR

Cestu ke konfiguračnímu souboru aplikace přijímá prostřednictvím parametru „-o“ a nebo ho hledá ve výchozí cestě, konkrétně v „/etc/collector/collector.conf“.

```
# comment
directive_name value
```

Listing 4.2: Syntaxe konfiguračního souboru Kolektoru

4.1.3 Konfigurační aplikace

Spolu s Kolektorem poskytuji aplikaci naprogramovanou v jazyku Java, která spravuje jeho databázi. Tato aplikace zpracovává JMS zprávy, které konfigurují, jaké informace je třeba sbírat a kam jejich data zasílat. Aplikace přijímá komplexní JSON strukturu, kterou jsem zmínil v kapitole Návrh. Struktura konfiguruje veškeré informace ohledně hosta a data, která poskytuje. Obsahuje množinu OID včetně jejich případných rodičovských závislostí pro sběr.

Dále tato aplikace přijímá zprávy typu „REGISTER_OUTPUT_SERVICE“, které obsahují množinu itemů a nastavení pro jejich zasílání. Pokud v této množině přijde item, který není dosud v databázi, je nutné ho ponechat ve speciální tabulce a vyčkat než informace o tomto itemu přijdou z nadřazeného systému, případně od uživatele.

Aplikace má tedy přímočarý úkol. Poslouchat příchod JMS zpráv a na základě jejich tvaru upravovat relační databázi Kolektoru.

4.1.4 Pomocné entity

V krátkosti zde popíši entity, které jsou důležité k běhu Kolektoru a nepopisují struktury s daty pro samotný sběr.

```
1  class Logging {
2  public:
3    ~Logging();
4    static Logging* getLogger();
5    void EMERGENCY_LOG (string msg);
6    ...
7    void DEBUG_LOG (string msg);
8  private:
9    Logging() {};
10   static Logging *instance;
11   void logIt(int priority, string msg);
12   const string priorities[8] = {"EMERGENCY",...};
13   pthread_mutex_t loggingMutex;
14 };
```

Listing 4.3: Třída Logging

Třída Logging poskytuje možnost logování na standardní výstup a do systémového syslogu. Protože je nutné mít tuto možnost v každé části programu, tak jsem se rozhodl tuto třídu implementovat jako tzv. jedináčka. Jedináček je třída, která udržuje referenci sama na sebe a je dostupná prostřednictvím veřejné a staticky definované metody „getLogger“. Třída loguje do syslogu pomocí knihovny syslog a s předdefinovanou „facility“ LOG_LOCAL7.

Jedna z dalších pomocných tříd udržuje konfiguraci samotného Kolektoru. Tato třída obsahuje strukturu „map“, která je klíčovaná direktivami, které jsou definované v konfiguračním souboru. Abychom ji mohli použít ze všech paralelních vláken, je také definovaná jako jedináček.

Další důležitá třída se jmenuje „DbCommunicator“. Poskytuje pomocí knihovny „libpqxx“ komunikaci s databází PostgreSQL. Tato třída obsahuje dvě základní metody, které jsou volány z metod ostatních. Jedna z nich provede příkaz bez výstupu a druhá do výstupního parametru uloží data vrácená databází. Protože je nutné většinu SQL příkazů používat více než jednou, jsou metody pojmenovány dle jejich využití, jako například „getAllHosts“, která vrátí všechny hosty pro sbírání. Metody, které mají funkci získání všech hostů, mají jako výstupní parametr samotnou frontu, do které hosty vloží. Vytvoření celé struktury hostů je relativně složité, abychom ho mohli provést jedním větším příkazem. Ten je detailněji okomentován přímo ve zdrojovém kódu. Ostatní metody se nazývají stylem „setHostsStatusFrom4To0“. Tato metoda je například volána po aktualizaci hostů v operační paměti.

Třída „SnmpCommunicator“ umožňuje vláknům získávat data z hostů. Disponuje hlavně jednou důležitou metodou, kterou zde popíši pseudokódem a dále okomentuji.

```
1  obtainedData getSnmpData(Host, OID, bulk) {
2    obtainedData;
3    run = true;
4    setCommunicationProperties(Host, OID, bulk);
```

```
 5     while (run) {
 6        getHostData()
 7           switch (type) {
 8              case bulk:
 9                 if(haveFullSubtree()) {
10                    run = false;
11                 } else {
12                    saveObtainedData();
13                    setNewOID();
14                 }
15                 break;
16              default:
17                 saveObtainedData();
18                 run = false;
19                 break;
20              }
21        freeRestOfData();
22     }
23     finishConnection();
24     return obtainedData;
25  }
```

Listing 4.4: Metoda getSnmpData pseudokód

Protože Kolektor poskytuje možnost získání dat pomocí bulk dotazů, musel jsem implementovat metodu, která získá celý podstrom. Pokud je podstrom dat příliš dlouhý, tak je nutné poslat další dotaz pro získání zbytku hodnot v rámci jednoho podstormu. Takže dlouhé podstromy jsou získávány pomocí více bulk dotazů a jejich výsledky jsou zřetězeny do spojového seznamu, který metoda vrátí. Tato metoda otevře před získáváním dat relaci s příslušným hostem a na konci ji uzavře. Největší otázkou v této části bylo, jestli má smysl data získaná pomocí SNMP ukládat do vlastní struktury, což by znamenalo jejich kopírování a tudíž nepatrné zdržení, které by se při velkém množství hostů mohlo značně projevit. Jako další možnost se nabízelo použít strukturu, která je součástí použité knihovny Net-SNMP. Nakonec jsem se rozhodl použít strukturu z knihovny, která obsahuje další data, jako jsou datové typy a další informace o OID. Tato struktura představuje jednosměrně zřetězený seznam, který v každém uzlu obsahuje OID se získanou hodnotou, nazývá se „variable_list“.

Poslední podpůrnou třídou je třída s názvem „RabbitmqCommunicator“ reprezentující připojení do služby RabbitMq. Pro každé paralelní vlákno zpracovávající data existuje jedna instance této třídy. Nastavení připojení je získáno z konfiguračního souboru a parametry pro odeslání dat jsou získány z databáze, abychom měli možnost nastavit parametry během spuštěného Kolektoru. Poskytuje pouze metodu „sendMessage“, která přijímá data pro zaslaní v podobě řetězce a zmíněné parametry.

4.1.5 Důležité entity

Entity v této sekci popisují data pro sbírání a pro jejich uložení. Tato množina entit je velmi provázaná. Entity zde postupně popíši včetně jejich účelu a později představím vztahy mezi nimi.

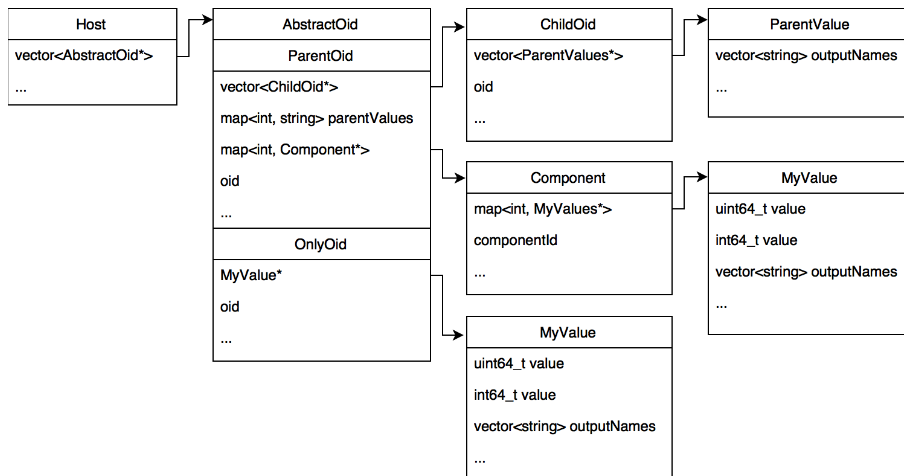
Nejdůležitější je samotný „Host“, ten obsahuje informace, které jsou nutné pro konfiguraci SNMP komunikace, jako je podporovaná verze SNMP, community v případě verze 2c a v případě verze 3 další parametry. Také je zde uložen jeho identifikátor získaný z databáze, který se použije pro vytvoření názvu metriky. Protože někteří hosté jsou pomalí při generování jejich odpovědi, bylo nutné poskytnout administrátorovi možnost nastavit timeout SNMP komunikace pro každého hosta zvlášť, takže zde nalezneme i tento parametr.

Pro větší přehlednost a jednoduší programování jsem použil abstraktní třídu, která popisuje obecný tvar OID určeného ke sběru. Tato třída se jmenuje „AbstractOid“ a obsahuje základní vlastnosti OID, hlavně jméno, datový typ a samotné OID v podobě řetězce. Třída popisuje několik důležitých virtuálních metod, které se liší svou implementací pro různé typy sběru. Jedna z nich je metoda „getData“, která získá data přes SNMP pomocí daného OID. Druhá metoda je „sendData“, která data zřetězí a připraví pro odeslání.

Ze třídy „AbstractOid“ dědí dvě třídy, které implementují zmíněné metody. Jedna ze tříd se jmenuje „OnlyOid“, tato třída implementuje sběr OID, které se nesbírají pomocí bulk dotazu, ale pouze dotazem typu GET, tzn. obsahuje OID, které odkazuje přímo na hodnotu. V této třídě najdeme hlavně samotné OID, jeho aktuální získanou hodnotu a seznam názvů parametrů, pomocí kterých chceme data odeslat. Implementace zmíněných metod je relativně triviální. Metoda „getData“ pomocí třídy „SnmpCommunicator“ získá hodnotu, kterou zkontroluje a dle datového typu si ji uloží. Stejně tak metoda „sendData“ pouze zkontroluje, jestli data chceme odeslat do příslušné služby a vrátí jejich zřetězenou podobu. Druhá třída, která dědí parametry ze třídy „AbstractOid“ je „ParentOid“. A jak název napovídá, bude obsluhovat bulk dotazy, pro které je nutné nejprve získat hodnoty z tzv. rodičovského OID a dle nich rozhodnout, která data uložit a použít. Tuto vlastnost jsem popsal v kapitole Návrh. Třída je úzce spjata s třídou „Component“ udržující hodnoty rodičovského OID určené pro další zpracování. Implementace metody „getData“ se na první pohled jeví jako mnohem složitější, ale ve své podstatě je rozdělena jen na dvě dost podobné části. První část získá data z rodičovského OID, která uloží do mapy klíčované indexem z OID. V druhé části se postupně získávají data z OID, které zde označuji jako „ChildOid“ a na základě hodnot rodičovského OID, pro které chceme data zpracovávat, je uloží. Pro toto uložení „ParentOid“ udržuje mapu „Component“, která obsahuje samotná získaná data.

Již zmíněná třída „ChildOid“ obsahuje množinu hodnot rodičovského OID, které chceme zpracovat. Kromě těchto hodnot třída obsahuje i informace o OID, jako je jeho datový typ a jméno. Hodnoty rodičovského OID jsou uloženy prostřednictvím struktury nazvané „ParentValue“, která obsahuje seznam cílových služeb, do kterých jsou data odeslána.

Protože data mají několik datových typů a je nutné je ukládat v různém tvaru, tak jsem se rozhodl využít vlastní strukturu pro reprezentaci samotné hodnoty. Tato třída má název „MyValue“ a obsahuje informace nutné pro další zpracování. Hodnoty získané prostřednictvím SNMP můžou být například datového typu COUNTER64, pro který potřebujeme využít ekvivalentně velký datový typ v programovacím jazyku. Třída také obsahuje časovou značku, ve které je čas získání dat. Protože pro následné zpracování potřebujeme informaci o cílových službách, udržuje i ukazatel na množinu jejich názvů.



Obrázek 4.2: Vztahy nejdůležitějších datových struktur

Obrázek č. 4.2 zobrazuje přehled nejdůležitějších vztahů dříve představených tříd.

4.1.6 Hlavní vlákno

```
1  mainThread() {
2    readInputParameters()
3    readConfigFile()
4    createThreads()
5    waitingHostsQueue <- getHostsForCollecting()
6    timeToRun <- getNextRunTime()
7    while(true) {
8      while(true) {
9         if(timeToRun or endInstance)
10          break;
11        sleep()
12      }
13      if(endInstance)
14        break
15      hostsForDelete <- readHostsForUpdateAndDelete()
16      for host in waitingHostsQueue {
17        if(host in hostsForDelete)
18          delete host
19        else
20          collectingHostsQueue.put(host)
21      }
22      newHosts <- getNewHostsForCollecting()
23      for host in newHosts
24        waitingHostsQueue.put(host)
25      timeToRun <- getTimeOfNextRun()
```

```
26     }
27     waitForThreads()
28     freeMemory()
29 }
```

Listing 4.5: Torzo hlavního vlákna

Hlavní vlákno se při spuštění postará o načtení konfiguračního souboru, o vytvoření ostatních vláken a poté zůstane v nekonečné smyčce, ve které na základě spočítaného času vloží hosty do fronty pro sbírání. Po vložení hostů pro sběr zjistí jestli je v databázi nějaký host, který se aktualizoval, a vloží ho do fronty, ve které čeká na další iteraci. Před každým spuštěním vlákno zkontroluje, jestli uživatel nenastavil pomocí signálu hodnotu proměnné „endInstance“ a pokud ano, tak vlákno počká na ukončení dříve spuštěných vláken a poté se ukončí.

4.1.7 Sbírací vlákno

```
1 collectingThread() {
2   while(true) {
3     if(endInstance)
4       break
5     host <- collectingHosts.get()
6     host.getData()
7     processingDataQueue.put(host)
8   }
9 }
```

Listing 4.6: Torzo vlákna sbírajícího data

Torzo zjednodušeně popisující funkcionalitu vlákna, které sbírá data, je krátké a skládá se především ze získání hosta, jehož data bude sbírat. Po získání hosta sebere jeho data a vloží ho do fronty pro další zpracování. Stejně jako předešlé vlákno i toto kontroluje, jestli není konec instance programu a pokud ano ukončí se.

4.1.8 Zpracovávajicí vlákno

```
1 processingDataThread() {
2   outputServices <- getOutputServicesFromDb()
3   while(true) {
4     if(endInstance)
5       break
6     host <- processingDataQueue.get()
7     for outputservice in outputServices {
8       dataForSend <- host.getDataForSend(outputservice)
9       outputservice.sendData(dataForSend)
10    }
11    host.freeData()
12    waitingQueue.put(host)
13  }
14 }
```

Listing 4.7: Torzo vlákna zpracovávajícího hosty

Třetí a poslední typ vlákna, které se stará o odesílání dat do příslušných služeb, opět čeká na svého hosta a když ho dostane, zřetězí všechna jeho data, která mají informaci o odeslání do aktuální služby. Pokud jsou data určena tomuto zdroji, tak jsou odeslaná a po zpracování všech cílových služeb jsou hodnoty u hosta smazány z operační paměti. Po uvolnění prostoru v paměti je host předán do fronty, ve které čeká na další iteraci sběru.

4.1.9 Instalace a kompilace

Vývoj Kolektoru probíhal na systému Debian GNU/Linux ve verzi Jessie, který používá pro instalaci softwaru balíčkovací systém apt. Rozhodl jsem se tedy vytvořit balíček pro instalaci této komponenty. Balíček obsahuje zkompilovaný program, takže je závislý na architektuře, na které ho lze použít.

Pro kompilaci Kolektoru jsem použil nástroj Make, který používá soubor Makefile s popisem kompilace a následného slinkování. Pro kompilaci i slinkování jsem použil nástroj g++ a jeho výchozí linker. Tento překladač používám s několika přepínači, jako jsou například „pedantic“ a „Wall“, které pomáhají při hledání chyb a kontrolují správné použití některých konstrukcí v jazyku C++. Pro slinkování je nutné použít dodatečné knihovny, které jsou lpqxx, lsnmp, lpthread a lrabbitmq. Tyto knihovny poskytují komunikaci prostřednictvím SNMP, komunikaci s databází, komunikaci s RabbitMq a vytvoření vláken typu posix. Pro samotný překlad používám občas trochu kontroverzně vnímaný přepínač „std=c++11“, který spustí kompilaci ve standardu C++11.

4.2 Ukládání dat a generování grafů

4.2.1 Instalace

Protože Graphite prochází stále progresivním vývojem, nemohl jsem použít verzi, která je dostupná ve výchozím repozitáři operačního systému Debian Jessie, na kterém jsem aplikaci spouštěl. Dohledal jsem tedy nejnovější stabilní verzi, což byla 0.9.15, kterou jsem nainstaloval a použil. Graphite je distribuovaný pomocí balíčků pro všechny běžné balíčkovací systémy včetně „apt“.

4.2.2 Nastavení parametrů

Po instalaci stačí nakonfigurovat několik direktiv v konfiguračním souboru, který je typicky uložený v adresáři „/etc/carbon“. Tento soubor obsahuje většinu konfigurace pro správný běh, jak jsem popsal v kapitole Analýza. Pro požadované využití jsem nastavil správný přístup k AMQP získaný po nakonfigurování aplikace RabbitMq. Po testu příjmu dat bylo nutné změnit další direktivy, které ovlivňují rychlost a parametry ukládání dat. Prvním problémem bylo vytvoření velkého množství souborů, což lze ovlivnit parametrem, který je ve výchozím stavu nastaven na relativně malou hodnotu, aby aplikace nezahltila úložiště. Pro prvních několik iterací sběru dat prostřednictvím Kolektoru jsem musel tento parametr zvýšit a poté ho z důvodu bezpečnosti znovu vrátit na původní hodnotu. Protože je aplikace v mém případě spuštěna na perzistentním úložišti typu SSD, tak jsem změnil i parametr, který určuje pořadí ukládání dat. Protože náhodné zápisy pro HDD jsou většinou kritické z pohledu výkonu, snaží se to Graphite ovlivňovat pomocí mechanizmů řadících data pro uložení do kompaktních celků. Ale protože SSD není náchylné

na náhodné zápisy, tak jsem změnil i tento parametr, který dovoluje Graphitu ukládat data, tak jak jsou na řadě a nijak je nemusí řadit.

4.2.3 Předpis souborů pro data

Aby uživatel mohl získat grafy metrik, které určil pro sbírání prostřednictvím Kolektoru, tak jsem určil syntaxi jejich názvů, kterou jsem představil v kapitole Návrh. Díky této syntaxi je uživatel schopen data dohledat a použít.

Řekněme, že do databáze kolektoru jsme nastavili následující parametry pro sbírání a nastavili jako cílovou službu Graphite:

```
Host: id: 1, hostname: 'switch1'
Component: id: 1, name: 'GigabitEthernet0/1'
Oid: id: 1, name: 'ifOutErrors'
```

Potom příslušnou metriku, kterou můžeme vykreslit do grafu, nalezneme pod názvem „1.1.1“. Pro testovací účely a pro představu, jak je možné s daty manipulovat, jsem nakonfiguroval i aplikaci Grafana.

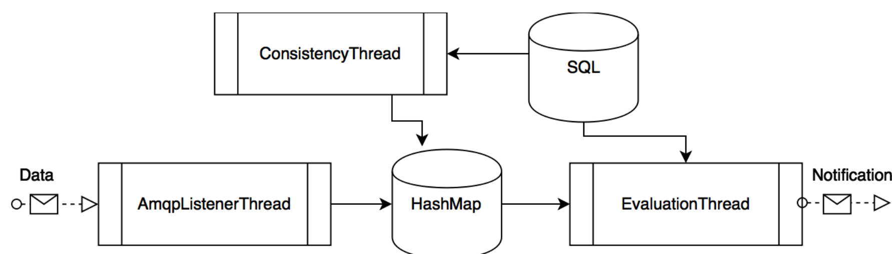
4.3 Vyhodnocování dat

Vyhodnocování dat jsem naprogramoval v jazyku Java, protože poskytuje lepší prostředky pro práci s použitými službami. Tuto vlastnost jsem ocenil, protože jsem musel využít relativně velké množství externích knihoven, případně dalších frameworků. Po vyhodnocení situace, kdy jsem zvažoval použití nějaké existující databáze, která udržuje data v operační paměti, jsem se rozhodl využít implementaci hašované mapy se jménem „ConcurrentHashMap“, která umožňuje bezpečný přístup prostřednictvím více pracujících vláken. V této mapě udržuji data, která přichází prostřednictvím AMQP. Pro konfiguraci jsem použil komunikační kanál JMS, který používá formát dat JSON. Tato data konfigurují výrazy, které má program kontrolovat a vyhodnocovat. Část těchto informací je nutné získat prostřednictvím komunikace s Kolektorem. Kolektoru je také nutné sdělit informaci o zasílání dat na příslušné místo v AMQP službě.

Implementace je realizovaná jako Webová aplikace, kterou je možné spustit v jednom z mnoha servlet kontejnerů a je distribuovaná v podobě balíčku typu „war“. Při vývoji jsem využil kontejner serveru Apache Tomcat verze 8. Aplikace používá jeden z nejznámějších frameworků Spring. Ten poskytuje velké množství možností, ale jeho hlavní výhodou je správa inicializací tříd v rámci jedné instance [32]. Pro správu instancí z relační databáze jsem použil framework Hibernate, který zjednodušuje správu dat v databázi [33].

4.3.1 Architektura

Architektura je zobrazena na obrázku 4.3. Na obrázku jsou dvě různá úložiště. První z nich je HashMap, což je již zmíněná databáze pro přijatá data. Relační databáze, která udržuje dohledové služby včetně informací o zprávách zasílaných uživatelům. Šipky na obrázku určují směr toku dat z jednotlivých vláken. Na obrázku je znázorněno, že například vlákno „ConsistencyThread“ předává nastavení dat do operační paměti. Toto vlákno se stará v první řadě o nastavení počtu historických dat a intervalu ukládaných dat. Vlákno „AmqpListener“



Obrázek 4.3: Architektura komponenty vyhodnocující data

pouze naslouchá na příslušné frontě ve službě RabbitMq a přijímá data, která ukládá do databáze v operační paměti. Z výkonnostních důvodů je toto vlákno odděleno od komunikace s relační databází. Je důležité mít možnost zpracovat velké množství nárazově přijatých dat. Další a zároveň nejdůležitější vlákno, které lze na základě konfigurace spustit ve více instancí, vyhodnocuje zmíněné výrazy a generuje notifikace. Toto vlákno získává data o triggerech pro vyhodnocení z relační databáze a data pro jejich vyhodnocení z mapy. Při stavu triggeru, který je určen pro odeslání, odešle zprávu typu JMS s uživateli, kteří jsou v příslušných skupinách.

Data ukládaná do mapy obsahují zarovnané časové značky, které jsou vytvořeny pomocí intervalu příchodu dat. K tomuto kroku jsem se rozhodl hlavně z důvodu časové náročnosti nalezení správné historické hodnoty pro příslušný výraz. Díky této vlastnosti a s pomocí několika hašovacích map jsou data nalezena v konstantní časové složitosti.

4.3.2 Konfigurace

Aplikace se konfiguruje prostřednictvím konfiguračního souboru. Tento konfigurační soubor obsahuje hlavně konfiguraci použitých služeb. Konfiguruje se zde připojení do služby RabbitMq vyžadující ve většině případů ověření, název fronty pro data a název exchange, který je spojen s použitou frontou. Pro komplexnější komunikaci je možné použít směrovací klíč, ale ten není ve většině případů nutný. Dále se zde konfigurují informace o službě, která přijímá notifikační zprávy pro zpracování a pro jejich odeslání ke koncovým uživatelům. To se skládá hlavně z nastavení komunikace a cílové fronty. Nejdůležitější parametr je pravděpodobně ten, který určuje počet vláken, které vyhodnocují triggery. Protože knihovna použitá pro komunikaci s RabbitMq umožňuje paralelní čtení z příslušné fronty, najdeme tento parametr též v konfiguračním souboru.

4.3.3 Entity

Jak jsem zmínil v úvodu kapitoly, tak v aplikaci existují tři nejdůležitější entity. Entita Trigger, Item a MonitoringService. Tyto entity jsou vzájemně provázané vztahem „n:n“. Tento vztah je důležitý, protože je nutné některé triggery případně itemy sdílet mezi více službami. Řekněme, že uživatel vytvoří službu, která kontroluje například stav konkrétního rozhraní a stejně tak vy-

tvoří službu, která kontroluje stav všech rozhraní na daném síťovém prvku. Potom jsme v situaci, kdy je nutné itemy a triggery sdílet mezi oběma službami. Důvodem, proč jsem se rozhodl použít pro konfiguraci jednotku, jako je služba, a ne konkrétního hosta, je hlavně fakt, že podoba konfigurace bude přívětivější pro uživatele. Při správné implementaci mohou zmíněné služby sloužit jako šablony pro konfiguraci dohledovaných hodnot. Entita „MonitoringService“ udržuje hlavně vazbu mezi itemy a triggery a momentálně neobsahuje kromě názvu žádnou další přidanou informaci. Slouží hlavně jako konfigurační jednotka a budu se jí věnovat v následující podkapitole o konfiguraci. Entita trigger, která popisuje samotné výrazy, které je nutné vyhodnocovat, je používaná vláknem, které je k tomu určené. Tato entita obsahuje vazbu na textové řetězce, které se odesílají uživateli při informování o změnách stavu. Trigger prochází v životním během své existence několika stavy:

NODATA stav, který říká, že v operační paměti není dostatek dat pro vyhodnocení výrazu triggeru

UNKNOWN stav, ve kterém se trigger nachází, pokud je právě přidán

TRUE poslední vyhodnocení výrazu mělo pozitivní výsledek

FALSE poslední vyhodnocení výrazu mělo negativní výsledek

Nejdůležitějším atributem každého triggeru je již několikrát zmíněný výraz, který je vyhodnocován. Jeho syntaxe se přímo odvíjí od knihovny, kterou jsem použil, a od funkcí, které poskytuje. Protože výraz je reprezentovaný řetězcem, který obsahuje přímou vazbu na item prostřednictvím jeho názvu. Tak lze dohledat potřebná data a toho využívá vyhodnocovací vlákno.

```
${1.1.1.last(0)} > ${1.1.1.last(1)}
MIN(${1.1.1.last(0)}, ${1.1.1.last(1)}, ${1.1.1.last(2)}) > 100
${1.1.1.last(0)} * 10 > MAX(${1.1.1.last(1)}, ${1.1.1.last(2)})
```

První výraz kontroluje, jestli je aktuální hodnota větší než hodnota předchozí, což je například trigger, který kontroluje, jestli roste počítadlo chyb na rozhraní síťového zařízení. Další výraz kontroluje, jestli je minimum ze tří předešlých hodnot větší než hodntoa 100 a poslední kontroluje, jestli je desetinásobek aktuální hodnoty větší než maximum ze dvou předešlých hodnot.

Hodnota itemu se vždy vloží za řetězec ve tvaru „${název.itemu.funkce(index v historii)}“. V současné verzi je pro předzpracování dat dostupná pouze funkce „last“, která vloží hodnotu z databáze beze změn. Funkcí, které jsou použitelné pro vyhodnocení samotných dat, je celá řada a jsou popsány v [34].

Entita „Item“ obsahuje především interval, který určuje čas mezi příchozími daty. Tento interval je důležitý hlavně pro zarovnání časové značky. Dalším parametrem je počet historických dat, která jsou potřeba pro vyhodnocení triggerů. Tyto dva parametry jsou synchronizovány do databáze v operační paměti pomocí vlákna „ConsistencyThread“. Pro ulehčení konfigurace jsem se rozhodl využít informace o itemech, které jsou v Kolektoru. Takže uživatel definuje jen triggery u příslušné služby a tato komponenta si prostřednictvím Kolektoru zjistí další informace.

Při přijetí triggeru s neexistujícím itemem je nutné ho vytvořit a označit, že je nutné k němu získat další informace a nemá se používat pro vyhodnocení výrazů. Na základě vytvoření neexistujícího itemu je vygenerována zpráva pro Kolektor, která ho žádá o dodatečné informace o itemu. Item, který neobsahuje všechny nutné informace, je ve stavu označeném „NEW“ a po jejich získání je ve stavu „READY“.

Pro ukládání itemu s jeho daty do hašovací mapy jsem vytvořil entitu „ItemData“, která udržuje jeho nejdůležitější parametry, a to hlavně ty, které jsou potřebné pro vyhodnocení jeho výrazů. Instance tohoto typu obsahuje informaci o počtu historických dat a o délce intervalu. Tyto parametry využívá prostřednictvím funkce „clearData()“ odstraňující historická data, která už nejsou potřeba a dle parametrů by neměla být v databázi. Tato funkce se volá vždy když vkládáme hodnoty do databáze.

4.3.4 Vyhodnocovací vlákno

Nejdůležitějším vláknem, které může běžet v několika instancích, je vlákno vyhodnocující výrazy triggerů. Protože každý výraz je nutné v každém časovém intervalu vyhodnotit právě jednou, je nutné při větší množině dat rozdělit triggery mezi všechna vlákna. Toto rozdělení probíhá už při vybírání triggerů z databáze a to pomocí operátoru modulo, identifikátoru vlákna a počtu vláken. Trigger se zařadí do množiny konkrétního vlákna pouze tehdy, pokud jeho identifikátor vyhovuje následující formuli:

$$(trigger_id\%number_of_threads) = thread_number$$

Díky této formuli lze jednoduše rozdělit triggery téměř rovnoměrně mezi všechna vlákna. Tento způsob spoléhá na situaci, kdy nedochází příliš často k obměně triggerů a tedy ke generování jejich nových identifikátorů, protože by snadno mohlo nastat přetížení některého z vláken. Pokud taková situace nastane, je nutné hodnoty znovu vygenerovat.

Toto vlákno dále prochází svou množinu triggerů, kterou se před každým během snaží obohatit o nové triggery z databáze. Při průchodu triggerů zpracuje jejich výrazy tak, že je rozdělí na jednotlivé části jako funkce, název itemu a parametr funkce, který určuje jak starou hodnotu potřebujeme pro vyhodnocení. Pro získání historické hodnoty je nutné si z aktuálního času vypočítat zarovnaný čas aktuální hodnoty a od ni odečíst případný posun do historie. Dále jen získat hodnotu z mapy s daty itemu. Poté nahradí proměnné ve výrazu triggeru za konkrétní hodnoty a předá ho k vyhodnocení. Podle výsledku vyhodnocení určí, zda je nutné zaslat zprávu a uložit nový status triggeru do databáze.

4.3.5 Konfigurace pomocí JMS

Pro konfiguraci příslušné dohledové služby slouží struktura dat přijatá pomocí JMS. Struktura obsahuje v první části informaci o události, které se zpráva týká, v tomto případě „MONITORING_SERVICE“, a o operaci, která může být „ADD“ nebo „REMOVE“ pro přidání čí odebrání dohledové služby. Příklad této zprávy je zobrazený v příloze B.4. Obsahuje množinu triggerů, které jsou postupně zpracovány, provázány s itemy a uloženy do databáze. Historie

těchto itemů je nastavena na základě parametru funkce ve výrazu triggeru. Pro vyhodnocení výrazů v příkladu je nutné udržovat jednu historickou hodnotu. Pokud jde o zatím neexistující itemy, vygeneruje se zpráva pro Kolektor ve tvaru, který je zobrazen v příloze B.2. Struktura obsahuje jména itemů, které mají být zasílány z Kolektoru do dohledové části. V příkladu je vidět, že v Kolektoru se pro každou službu, která data přijímá, nastavují parametry pro posílání dat pomocí AMQP. Při odpovědi přichází itemy v obdobném tvaru viz B.3, ale s nastavenými parametry „step“, tedy intervalu sběru a datového typu v podobě parametru „sourceDataType“. Vytvoření skupin uživatelů, kterým se posílají notifikace probíhá prostřednictvím JMS, které musí být ve tvaru zobrazeném v příloze B.5. Při příchodu požadavku pro smazání skupiny uživatelů, která má stále vazby na notifikace je nutné nemazat tuto skupinu, aby v databázi nezůstal trigger, který nemá žádné vazby na skupiny. V tomto případě jsou ze skupiny smazáni pouze uživatelé a informace o pokusu smazání se zaloguje.

4.3.6 REST rozhraní

Pro poskytnutí výsledků vyhodnocených výrazů externímu nástroji jsem se rozhodl implementovat volání rozhraní REST poskytující triggery v určitém stavu. Uživatel si tak může jednoduše zobrazit triggery, které mají nedostatek dat a nebo triggery, které jsou v problémovém stavu. Data jsou poskytována ve formátu JSON a prostřednictvím URI končící řetězcem „/api/triggers“. Uživatel má možnost předat parametrem stav triggerů, o které má zájem. Z důvodu velkého množství triggerů jsem implementoval možnost stránkování ve výstupní množině triggerů. Pro stránkování je nutné vložit příslušné parametry.

Kapitola 5

Testování

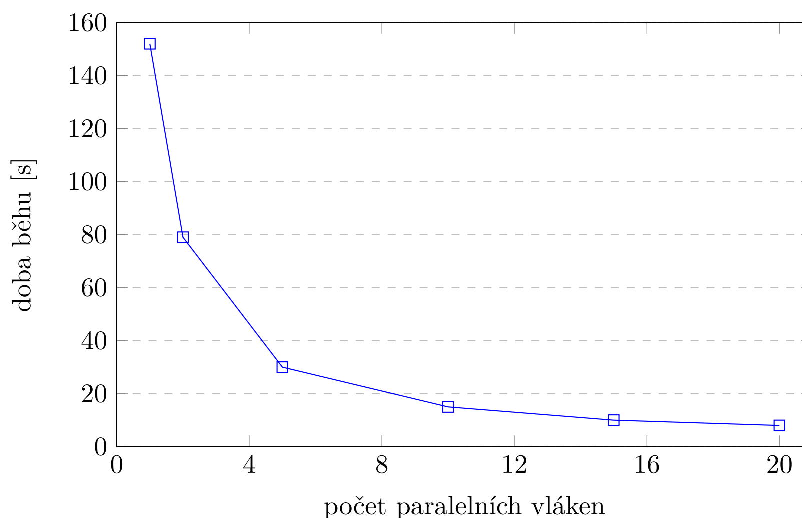
5.1 Testovací prostředí

Testování probíhalo v metropolitní síti poskytovatele připojení do Internetu. Tato síť obsahuje 863 zařízení, která jsou schopna poskytovat data prostřednictvím SNMP verze 2c. Valná většina těchto zařízení pochází od společnosti Cisco Systems, Inc a jsou to většinou zařízení typu směrovač a přepínač. Na těchto zařízeních je přes 200 000 hodnot, které má smysl generovat do grafu, a zhruba 60 000 hodnot ke kontrole. Tato zařízení jsou propojena pomocí metalických a optických spojů.

Pro běh naprogramovaných aplikací a pro testování aplikace pro generování grafů jsem využil server s operačním systémem Debian GNU/Linux ve verzi 8.7 na architektuře amd64. Server je plně virtualizovaný v prostředí od společnosti VMware. Konfigurace serveru je následující: 8GB operační paměti, 4 výpočetní jádra s rychlostí 2.20GHz. Server má k dispozici úložiště typu SSD o velikosti 25GB v diskovém poli RAID1. Do sítě je server připojen prostřednictvím virtuální síťové karty o rychlosti 1Gb/s.

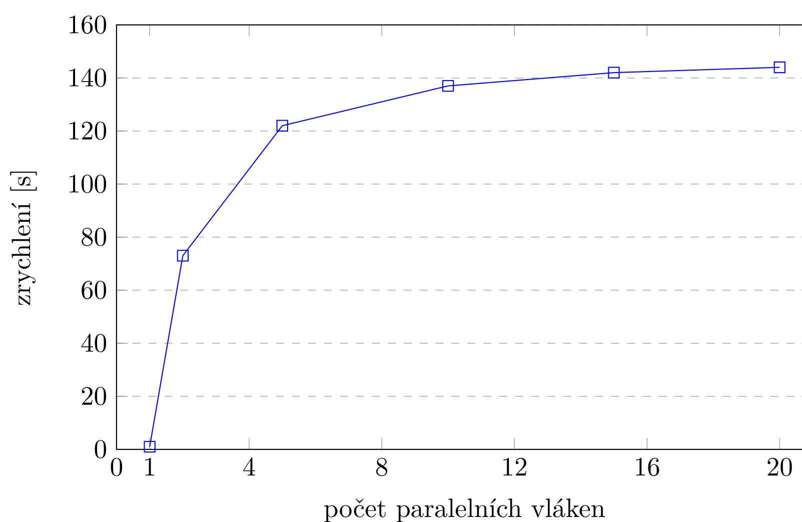
5.2 Kolektor

Testování Kolektoru probíhalo na již zmíněné síti a serveru. Testovací interval pro sbírání dat byl po celou dobu testování nastaven na 300s. Po získání všech dat vypíše Kolektor dobu iterace, kterou jsem použil pro vynesení do grafu. Pro testování jsem použil 863 prvků, na kterých jsem nakonfiguroval pomocí nástroje pro jejich správu SNMP verze 2c. Také bylo nutné povolit síťový provoz SNMP z testovacího serveru. Do databáze Kolektoru jsem vygeneroval konfiguraci sběru dat prvků. Dotazem GET přímo z prvků se sbírala data: uptime, využití procesoru a využití operační paměti. Pro testování bulk dotazů jsem využil síťová rozhraní prvků. Pro každé známé rozhraní jsem nastavil sbírání 13 OID, které měly stejné rodičovské OID. Hodnoty rodičovského OID obsahují názvy rozhraní, které jsem musel vložit do databáze. Pro sebrání těchto 13 hodnot z každého rozhraní se ideálně vygeneruje 14 dotazů typu bulk. Přebývající dotaz získá hodnoty rodičovského OID, které jsou nutné pro určení indexů jednotlivých komponent.



Obrázek 5.1: Doba běhu v závislosti na počtu paralelních vláken

V grafu je vynesena doba běhu jedné iterace Kolektoru pro různé počty paralelních vláken. Dle očekávání má funkce klesající tendenci s větším počtem paralelních vláken.



Obrázek 5.2: Zrychlení na základě počtu paralelních vláken

Na obrázku je vidět zrychlení v závislosti na počtu vláken, která jsou paralelně spuštěna. Zrychlení roste s logaritmickou funkcí, což je celkem dobrý předpoklad k využití Kolektoru pro větší prostředí.

Provedl jsem několik dalších testů, které kontrolovaly následující typy situací, které mohou nastat:

nedostupný prvek jeden z prvků je nedostupný – Kolektor zaloguje informaci a zkusí data získat při další iteraci

neexistující oid nastavení OID, pro které prvek neodpovídá – Kolektor zaloguje informaci se závažností „error“

práce s pamětí pomocí nástroje valgrind jsem zkontroloval uvolnění operační paměti po řádném ukončení

změna prvku změna konfigurace hosta během běhu Kolektoru – Kolektor hosta znovu načte a začne sbírat přidaná data

cppcheck pomocí nástroje cppcheck jsem kontroloval další možné problémy a chyby v použití konstrukcí jazyku

Škálování do šířky je možné pouze pomocí autonomních instancí kolektoru a jejich databází. Každý z těchto globálních Kolektorů může používat různý interval pro sběr dat a tím mimo jiné lze docílit častějšího sběru.

5.3 Generování grafů a ukládání dat

Před testováním nástroje Graphite jsem změnil hodnotu parametru, která omezuje maximální počet vytvořených souborů za sekundu. Touto změnou jsem docílil vytvoření všech souborů během několika prvních iterací.

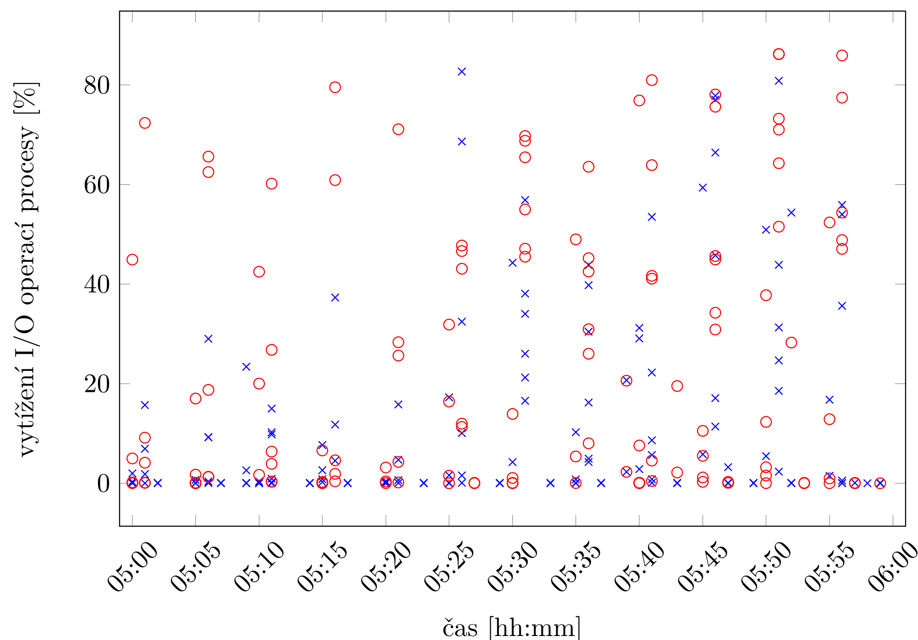
Dále jsem pro testování změnil parametr „MAX_UPDATE_PER_SECOND“ a parametr „MAX_CACHE_SIZE“. Toto jsou dva klíčové parametry, které ovlivňují velikost dočasné paměti pro data a maximální počet operací „whisper-update“ za sekundu. Po úpravě těchto parametrů a testování jsem docílil situace, že jsou data uložená během prvních několika sekund a využití procesů pro operace čtení a zápisu na perzistentní médium je na přijatelné úrovni, viz obrázek číslo 5.3. Z důvodu přehlednosti jsem do grafu nezanesl nulové hodnoty, protože jsou data sbírána každých 10s. Na grafu je zobrazené využití výpočetních jader pro operace zápisu a čtení na perzistentní médium. Data pro zápis přicházela každých 5 minut a první byla odeslána po sebrání v čase 5:00. Je zřejmé, že během prvních několika sekund až jedné minuty byla data zapsána.

Testování nástroje Graphite jsem provedl pomocí vytvořeného Kolektoru a všech hostů, kteří byli k dispozici. Pro testování jsem vytvořil obecný tvar databázových souborů, který je zobrazen v následujícím výpisu:

```
[default_average]
pattern = .*
xFilesFactor = 0.5
aggregationMethod = average

[default]
pattern = .*
retentions = 5m:7d,1h:31d,24h:1y
```

Pro agregaci se použije obecné pravidlo, které nastavuje agregační funkci „average“ a hodnota „xFileFactor“ umožňuje uložit agregovanou hodnotu i



Obrázek 5.3: Využití procesů pro I/O operace

v případě, že 50 % hodnot pro její výpočet chybí. Následující pravidlo konfiguruje, že se pro každou metriku vytvoří soubor s archivy, které mají retence nastavené dle direktivy „retentions". Pěti minutové hodnoty se budou ukládat na sedm dní a tak dále.

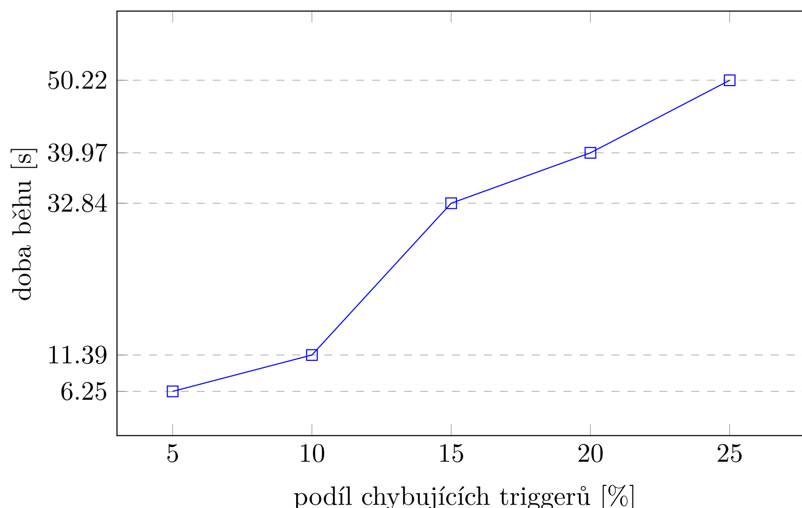
Databáze s tímto nastavením udržuje přesně 3150 hodnot, které jsou použité pro pozdější generování grafů. Databázový soubor v této konfiguraci má velikost 37552 B.

Pro tento test jsem spustil pouze démona „carbon-cache", který data přijímal ze služby RabbitMq, do které přicházela od zmíněného Kolektoru. Data ukládal do popsaných souborů a poskytoval k dalšímu zpracování pomocí API. Dobu, za jakou Graphite poskytl data po přijetí, jsem nebyl schopen změřit vzhledem k zanedbatelné prodlevě.

5.4 Vyhodnocování dat

Testování aplikace pro vyhodnocování výrazů předcházelo vytvoření krátkého skriptu v jazyku Bash, který odesílal data pro vyhodnocení. Tento skript byl nutný, abych mohl ovlivnit počet triggerů, které změní svůj stav během jedné iterace. Jak už jsem zmínil v části o implementaci, tak při změně stavu se vygeneruje příkaz „UPDATE" do relační databáze. Pro měření doby běhu iterace je tedy důležitý podíl triggerů, které změní svůj stav. Tento skript přijímal parametrem počet služeb a počet trigggerů pro službu, dobu iterace a procentuální podíl triggerů, které mění svůj stav při liché iteraci.

Pro vytvoření služeb, příslušných itemů a triggerů jsem použil dočasné



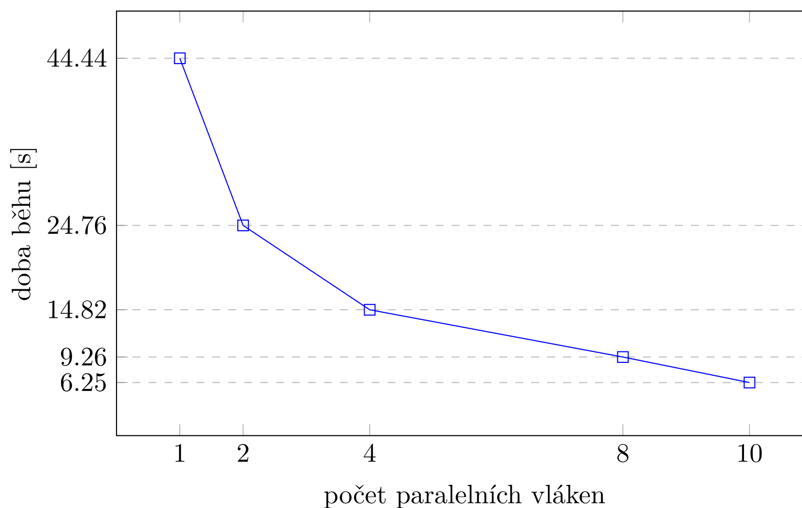
Obrázek 5.4: Podíl chybujících triggerů a doby běhu s 10 vlákny

REST volání, které vytvořilo potřebné objekty služeb a poslalo je do fronty ve službě ActiveMq, ve které si je stejná aplikace vyzvedla a zpracovala. Volání přijímalo parametry, které nastavily počet služeb a počet triggerů pro služby, které chceme vytvořit.

Skript i dočasné volání v API využívali stejnou jmennou konvenci. Jména itemů byla generována jako řada čísel (stejně jako při použití ID z Kolektoru). Item s názvem „1.2.3“ by patřil hostovi s identifikátorem 1, jeho komponentě s identifikátorem 2 a s OID s identifikátorem 3.

Všechna měření byla provedena až po ustálení stavů triggerů, protože jak je zřejmé už z definice této aplikace, tak počet historických hodnot po spuštění je nulový a musí tedy přijmout prvních několik iterací (v mém testovacím prostředí minimálně dvě) a vyhodnotit výrazy triggerů. Pro měření jsem použil interval 5 minut, aby bylo možné pohodlně změřit dobu běhu pro jedno výpočetní vlákno. Pro měření jsem použil 60 000 triggerů, které obsahovaly výraz „${item1.last(1)}>${item1.last(0)}“, takže porovnávaly poslední dvě hodnoty a pokud starší hodnota byla větší než aktuální, tak změnily stav a vygenerovaly příkaz „UPDATE“ do databáze.

Na obrázku 5.4 jsou vykresleny hodnoty, které jsem naměřil. Měřil jsem běh při 10 výpočetních vláknech, která zpracovávala zmíněných 60 000 triggerů a podíl těch, které měnily svůj stav, jsem měnil. Při změně 5 % tedy 3000 triggerů při jedné iteraci je doba výpočtu 6,25 sekundy. Situace, kdy se v jedné iteraci změní stav takového množství kontrolovaných hodnot, je velmi výjimečná, ale pro testování výkonnosti jsem podíl triggerů s měnícím se stavem zvýšil až na 25 %. Na obrázku je vidět, že i při takovém množství se dá očekávat relativně přijatelný čas vyhodnocení. Z měření vyplývá, že závislost doby vyhodnocení a počtu chybujících triggerů je lineární.



Obrázek 5.5: Závislost doby běhu na počtu výpočetních vláken s 5 % chybujících triggerů

Dalším typem měření bylo prokázání zrychlování výpočtu pro více výpočetních vláken. Pro 5 % měnících se triggerů jsem změřil čas pro různý počet paralelních vláken. Z obrázku 5.5 vyplývá, že použití jednoho vlákna pro takové množství triggerů je velmi neefektivní, a proto je vždy výhodné počet vláken zvýšit a pohybovat se s ním blízko hodnoty 10.

Horizontální škálování této aplikace je možné pouze spuštěním další instance s vlastní databází. Poté je nutné změnit názvy kanálů, na kterých přicházejí data ze zprostředkovatelů zpráv, a v komunikujících aplikacích správně určit, kam která data zasílat.

Závěr

Cílem této práce bylo vytvořit komponenty pro monitoring síťové infrastruktury a využít existující nástroj pro generování grafů. Komponenty bylo nutné naprogramovat od úplného začátku, takže včetně návrhu architektury, která musela poskytovat nějakou formu paralelismu pro běh při vysoké zátěži.

První komponenta s názvem Kolektor, která má za úkol data pro monitoring sbírat, vyžadovala implementaci další doplňující aplikace pro správu databáze. Kolektor je díky jeho flexibilní architektuře možné škálovat pomocí spuštění více výpočetních vláken nebo jej lze spustit ve více než jedné instanci. V takové situaci nebudou sdílet téměř nic a o rozdělení dat se musí postarat vyšší vrstva. Kolektor z důvodu výkonu a šetření šířky pásma využívá pro sběr většího množství hodnot z jednoho cíle dotazy typu bulk, které umožňují získat více hodnot v jedné odpovědi. Získaná data poskytuje pomocí AMQP, který je umožňuje přijímat do velké škály dalších nástrojů. Doplňující aplikace pouze přijímá konfigurační příkazy a upravuje jeho databázi.

Druhá komponenta přijímá data a pomocí uživatelem definovaných formulí vyhodnocuje, jestli vyhovují předepsaným výrazům. V této komponentě jsem navrhl jazyk pro formulaci vyhodnocovaných výrazů. Umožnil jsem také spuštění více než jednoho vlákna, které data vyhodnocuje. Přední výhodou je ukládání historických dat do operační paměti, a tak šetřit čas zápisu. Dále jsem navrhl jednoduchou konfiguraci dohledových služeb, které obsahují pouze triggery a příslušné itemy se vytvoří na základě jejich výrazů. Další informace nutné pro správný běh se získají od aplikace Kolektoru, která spravuje databázi. Zprávy, které jsou generované chybovými stavy výrazů, jsou posílané pomocí JMS, které umožňuje další zpracování téměř pro jakýkoliv účel.

Třetí komponenta, která má za cíl data perzistentně ukládat a poskytovat je pro generování grafů, umožňuje celou řadu možností, jak optimalizovat zatížení perzistentního média, včetně paralelního zpracování v podobě horizontálního škálování. Tato komponenta není nijak závislá na typu přijímaných dat, nerozděluje konkrétní hodnoty a ke všem přistupuje jako k číslu. Data poskytuje pomocí API pro další zpracování.

Cíle, které jsem si na začátku práce vytyčil, byly splněny a výhody, které tento systém poskytuje, co se týkají výkonnosti, jsem změřil a popsal v kapitole Testování.

Budoucí vývoj

Pro budoucí vývoj se nabízí celá řada možností, které může komponenta vyhodnocující data poskytnout. Jsou to především další možnosti vyhodnocovaných výrazů, jako jsou jejich závislosti napříč hodnotami. Určitě by se našla celá řada funkcí pro vyhodnocení výrazů, které lze vytvořit na míru konkrétnímu prostředí. Pro budoucí vývoj je připravena i možnost předzpracování dat před vyhodnocením samotného výrazu. V otázce optimalizace bych do budoucna uvažoval o přesunutí stavů triggerů také do operační paměti, aby bylo možné dosáhnout daleko větších výkonnostních možností, protože nejslabším místem je momentálně jejich úprava v databázi.

Vývoj Kolektoru bych směřoval hlavně do zpřehlednění zdrojového kódu a do jeho optimalizace hlavně v oblasti komunikace s databází. Dále se určitě nabízí možnost implementace plánovače sbírajících vláken, který bude umožňovat různé intervaly v rámci spuštěné instance.

Literatura

[1] Overview of Time Series Characteristics. 2017, [cit. 2017-04-08]. Dostupné z: https://onlinecourses.science.psu.edu/stat510/node/47

[2] Alonso, A. M.; García-Martos, C.: Time Series Analysis - Time series and stochastic processes. [online], 2012, [cit. 2017-04-03]. Dostupné z: http://www.etsii.upm.es/ingor/estadistica/Carol/TSAtema3petten.pdf

[3] RRDtool - Round Robin Database Tool. [online], 2006, [cit. 2017-03-26]. Dostupné z: https://www.caida.org/tools/utilities/rrdtool/

[4] Oetiker, T.: RRDtool Documentation. [online], 2012, [cit. 2017-04-03]. Dostupné z: http://oss.oetiker.ch/rrdtool/doc/index.en.html

[5] The Whisper Database. [online], 2016, [cit. 2017-04-05]. Dostupné z: http://graphite.readthedocs.io/en/latest/whisper.html

[6] Leinartas, M.: Whisper FAQ. [online], 2011, [cit. 2017-04-03]. Dostupné z: http://graphite.wikidot.com/whisper

[7] Trubetskoy, G.: Time Series Accuracy - Graphite vs RRDTool. [online], 2015, [cit. 2017-04-01]. Dostupné z: https://grisha.org/blog/2015/05/04/recording-time-series/

[8] Pratama, I.; Permanasari, A. E.; Ardiyanto, I.; aj.: A review of missing values handling methods on time-series data. In *2016 International Conference on Information Technology Systems and Innovation (ICITSI)*, Oct 2016, s. 1–6, doi:10.1109/ICITSI.2016.7858189.

[9] OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0. [online], October 2012, [cit. 2017-04-11]. Dostupné z: http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-complete-v1.0-os.pdf

[10] Ledbrook, P.: Understanding AMQP, the protocol used by RabbitMQ. [online], 2010, [cit. 2017-04-11]. Dostupné z: https://spring.io/blog/2010/06/14/understanding-amqp-the-protocol-used-by-rabbitmq/

[11] AMQP 0-9-1 Model Explained. [online], [cit. 2017-04-12]. Dostupné z: https://www.rabbitmq.com/tutorials/amqp-concepts.html

[12] Pivotal Software, I.: RabbitMq. [online], 2007, [cit. 2017-04-02]. Dostupné z: https://www.rabbitmq.com

[13] Mark Hapner, R. S., Rich Burridge: JMS is an API for accessing enterprise messaging systems from Java programs. [online], 1999, [cit. 2017-04-15]. Dostupné z: https://docs.oracle.com/cd/E19957-01/816-5904-10/816-5904-10.pdf

[14] ActiveMq. [online], 2011, [cit. 2017-04-18]. Dostupné z: http://activemq.apache.org

[15] Graphite Monitoring Architecture. [online], [cit. 2017-04-02]. Dostupné z: http://graphite.readthedocs.io/en/latest/_images/overview.png

[16] Project, T. G.: Graphite - Functions. [online], 2016, [cit. 2017-04-04]. Dostupné z: http://graphite.readthedocs.io/en/latest/functions.html

[17] Project, T. G.: Graphite - Documentation. [online], 2016, [cit. 2017-04-04]. Dostupné z: http://graphite.readthedocs.io/en/latest/overview.html

[18] Grafana Documentation. [online], 2017, [cit. 2017-03-25]. Dostupné z: http://docs.grafana.org

[19] Heard, C.: Guidelines for Authors and Reviewers of MIB Documents. [online], September 2005. Dostupné z: https://tools.ietf.org/html/rfc4181

[20] McCloghrie, K.: Management Information Base for Network Management. [online], May 1990. Dostupné z: https://tools.ietf.org/html/rfc1158

[21] OSI networking and system aspects – Abstract Syntax Notation One (ASN.1). [online], 2002. Dostupné z: https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf

[22] Case, J.: A Simple Network Management Protocol (SNMP). [online], May 1990. Dostupné z: https://www.ietf.org/rfc/rfc1157.txt

[23] Bruey, D.: SNMP: Simple? Network Management Protocol. [online], December 2005, [cit. 2017-03-26]. Dostupné z: http://www.rane.com/note161.html

[24] Aethis, U.: Security in SNMPv3 versus SNMPv1 or v2c. [online], 2002, [cit. 2017-04-06]. Dostupné z: https://www.scribd.com/document/48618460/snmpv3-vs-wp

[25] Case, J.: Protocol Operations for version 2 of the Simple Network Management Protocol. [online], April 1993. Dostupné z: https://tools.ietf.org/html/rfc1448

[26] Schmidt, D. R. M. . K. J.: *Essential SNMP*. O'Reilly, druhé vydání, 2005, ISBN 978-0-596-00840-6.

[27] Corbato, T. G. . S.: Comparing SNMPv2 with SNMPv1. [online], 1997, [cit. 2017-04-28]. Dostupné z: https://courses.cs.washington.edu/courses/csep561/97sp/paper1/paper14.txt

[28] Harrington, D.: An Architecture for Describing SNMP Management Frameworks. [online], January 1998. Dostupné z: https://tools.ietf.org/html/rfc2271

[29] Stallings, W.: [online], [cit. 2017-04-19]. Dostupné z: http://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-20/snmpv3.html

[30] Net-SNMP. [online], 2013, [cit. 2017-03-02]. Dostupné z: http://www.net-snmp.org

[31] Zabbix documentation 3.2. [online], 2017, [cit. 2017-04-07]. Dostupné z: https://www.zabbix.com/documentation/3.2

[32] Spring Framework. [online], [cit. 2017-03-05]. Dostupné z: https://projects.spring.io/spring-framework/

[33] Hibernate ORM. [online], [cit. 2017-03-05]. Dostupné z: http://hibernate.org/orm

[34] Klimaschewski, U.: EvalEx - Java Expression Evaluator. [online], [cit. 2017-03-03]. Dostupné z: https://github.com/uklimaschewski/EvalEx

Příloha A

Seznam použitých zkratek

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

CBC-DES Cipher Block Chaining-Data Encryption Standard

DNS Domain Name System

DS Data Source

HMAC Keyed-hash Message Authentication Code

HTTP Hypertext Transfer Protocol

ICMP Internet Control Message Protocol

IoT Internet of Things

IPv4 Internet Protocol v4

IPv6 Internet Protocol v6

JMS Java Message Service

LDAP Lightweight Directory Access Protocol

MIB Management Information Base

OID Object Identifier

PDU Protocol Data Unit

REST Representational state transfer

RRA Round-Robin Array

RRD Round-Robin Database

SNMP Simple Network Management Protocol

SQL Structured Query Language

SSH Secure Shell

TCP Transmission Control Protocol

URL Uniform Resource Locator

USM User Based Security Model

XFF X-File Factor

Příloha B

Další přílohy


```
{ "clazz":"JmsMessage",
  "event":"HOST_ADD",
  "operation":"ADD",
  "data": {
      "clazz":"HostInput",
      "id":1,
      "ip4":"192.168.1.1",
      "hostname":"Switch1",
      "snmpVersion":"SNMP_3",
      "snmpCommunity":"rocommunity",
      "snmpTimeout":1,
      "securityLevel":"authPriv", "securityName":"monitoring",
      "securityAuthKey":"AuthKey", "securityAuthProto":"SHA1",
      "securityPrivKey":"PrivKey", "securityPrivProto":"AES",
      "oids": [{
              "id":1,
              "oid":".1.3.6.1.2.1.1.3.0",
              "name":"sysUpTimeInstance",
              "dataSourceType":"TIMETICKS"
      }],
      "components": [{
              "id":1,
              "name":"GigabitEthernet0/0/0",
              "oids": [{
                      "id":2,
                      "oid":".1.3.6.1.2.1.31.1.1.1.7",
                      "name":"ifHCInUcastPkts",
                      "dataSourceType":"COUNTER64",
                      "parentId":3,
                      "parent": {
                          "id":3,
                          "oid":".1.3.6.1.2.1.2.2.1.2",
                          "name":"ifDescr",
                          "dataSourceType":"STRING"
}     }]   }]        }}
```


Listing B.1: Tvar JSON pro vytvoření hosta v Kolektoru

```
{
    "clazz":"JmsMessage",
    "event":"REGISTER_OUTPUT_SERVICE",
    "operation":"ADD",
    "data":{
        "clazz":"OutputServiceInput",
        "serviceName":"Monitoring",
        "ip4":"192.168.1.1",
        "exchagne":"monitoring",
        "routingKey":"",
        "itemInputs": [
            {
                "itemName":"1.1.1",
                "itemName":"1.2.1",
                "itemName":"1.3.1"
            }
        ]
    }
}
```


Listing B.2: Tvar JSON pro registraci příjmu dat od Kolektoru


```
{
    "clazz":"JmsMessage",
    "event":"MONITORING_ITEMS",
    "operation":"ADD",
    "data": {
        "clazz":"ItemListInput",
        "items": [{
                "name":"1.1.1",
                "dataSourceType":"INT",
                "step":300
            },
            {
                "name":"1.2.1",
                "dataSourceType":"INT",
                "step":300
            },
            {
                "name":"1.3.1",
                "dataSourceType":"INT",
                "step":300
            }
        ]
    }
}
```


Listing B.3: Tvar JSON pro nastavení itemů ve vyhodnocovací komponentě

```
{
  "clazz":"JmsMessage",
  "event":"MONITORING_SERVICE",
  "operation":"ADD",
  "data": {
    "clazz":"MonitoringServiceDto",
    "id": 1,
    "name":"Traffic monitoring",
    "triggers": [
    {
      "expression":"${1.1.1.last(0)}>${1.1.1.last(1)}",
        "notificationText": {
          "notificationText":"Traffic on Switch1 Eth1",
          "recoveryText":"No traffic on Switch1 Eth1"
        },
      "receiver": ["Network admins"]
    },{
      "expression":"${1.2.1.last(0)}>${1.2.1.last(1)}",
        "notificationText": {
          "notificationText":"Traffic on Switch1 Eth2",
          "recoveryText":"No traffic on Switch1 Eth2"
        },
      "receiver": ["Network admins"]
    },{
      "expression":"${1.3.1.last(0)}>${1.3.1.last(1)}",
        "notificationText": {
          "notificationText":"Traffic on Switch1 Eth3",
          "recoveryText":"No traffic on Switch1 Eth3"
        },
      "receiver": ["Network admins"]
    }]
  }
}
```

Listing B.4: Tvar JSON pro nastavení triggerů ve vyhodnocovací komponentě

```
{
  "clazz":"JmsMessage",
  "event":"USER_GROUP",
  "operation":"ADD",
  "data": {
    "clazz":"UserGroupInput",
    "userGroupName":"Administrators",
    "contacts": ["user1", "user2", "user3"]
  }
}
```

Listing B.5: Tvar JSON pro vytvoření skupiny uživatelů

Příloha C

Obsah přiloženého CD

```
readme.txt ........................... stručný popis obsahu CD
bin ....adresář se spustitelnou formou implementace a s balíčky webových aplikací
src
├── impl ............................ zdrojové kódy implementace
├── thesis ................. zdrojová forma práce ve formátu LaTeX
└── tests ...................... zdrojové kódy testovacích nástrojů
text ............................................... text práce
└── thesis.pdf ................... text práce ve formátu PDF
```