Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

StatsD HDR Histogram agent

PCP Performance Metric Domain Agent pro StatsD protokol

2023

Bc. Miroslav Foltýn

Vedoucí práce: Mgr. Jan Outrata, Ph.D.

Studijní obor: Informatika, prezenční forma

Bibliografické údaje

Autor:	Bc. Miroslav Foltýn
Název práce:	StatsD HDR Histogram agent (PCP Performance Metric Domain Agent pro StatsD protokol)
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2023
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	Mgr. Jan Outrata, Ph.D.
Počet stran:	63
Přílohy:	1 DVD
Jazyk práce:	český

Bibliograhic info

Author:	Bc. Miroslav Foltýn
Title:	StatsD HDR Histogram agent (Performance Co-Pilot Performance Metric Domain Agent for StatsD protocol)
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2023
Study field:	Computer Science, full-time form
Supervisor:	Mgr. Jan Outrata, Ph.D.
Page count:	63
Supplements:	1 DVD
Thesis language:	Czech

Anotace

Obsahem práce je vytvoření konfigurovatelného agenta pro Performance Co-Pilot v jazyce C, který by zpracovával síťové zprávy StatsD protokolu reprezentující metriky různých typů, jejich hodnoty a další metadata. Agent bude tato data agregovat, hodnoty sbírat a díky integraci s Performance Co-Pilot je bude zpřístupňovat dalším utilitám umožňujícím inspekci, logování a monitoring. Součástí práce jsou integrační testy.

Synopsis

The aim of this thesis is to create a configurable agent for the Performance Co-Pilot in language C, that would process network messages of StatsD protocol representing metrics or various types, their values and metadata. Agent would aggregate this data, collect values, and thanks to integration with Performance Co-Pilot, make them available for further inspection, logging, and monitoring. This work includes integration tests.

Klíčová slova: Performance Co-Pilot; HDR Histogram; Jazyk C; Ragel; StatsD protokol; Bash; Make; Monitoring a sběr dat, Linux

Keywords: Performance Co-Pilot; HDR Histogram; C Language; Ragel, StatsD protocol; Bash; Make; Performance metric monitoring and collecting, Linux

Děkuji Mgr. Janu Outratovi, Ph.D., vedoucímu této práce, za odborné konzultace a Lukáši Zapletalovi za pravidelné konzultace a feedback. Velmi děkuji komunitě Performance Co-Pilot, hlavně Nathanu Scottovi, za adresaci mých otázek, za pomoc při střetu s problémy a laskavý přístup.

I would like to thank Jan Outrata, supervisor of this work, for consultations, and Lukáš Zapletal for regular consultations and feedback. Lastly I would like to thank the Performance Co-Pilot community, mainly Nathan Scott, for answering all my questions, being of help when I got stuck, and their kind approach.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

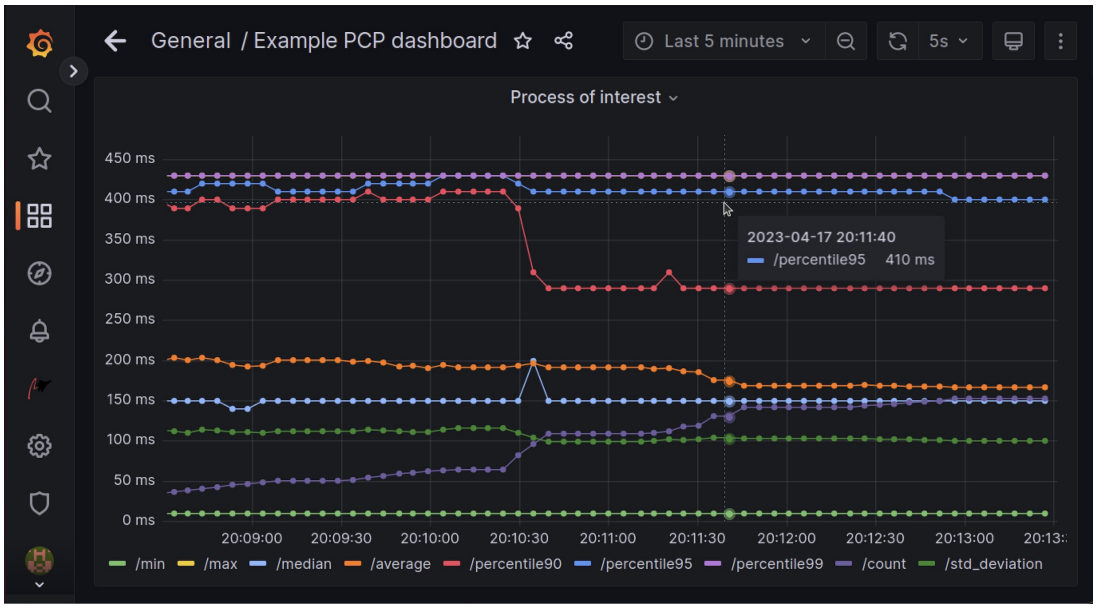
Seznam zdrojových kódů

1 Performance Co-Pilot

Performance Co-Pilot (dále jen 'PCP') je sada služeb a nástrojů, které umožňují analýzu operačních systémů, případně aplikací, prostřednictvím metrik z různých domén zájmu. PCP reprezentuje abstrakci pro kolekci metrik a práci s nimi. PCP samotný je napsán v jazyce C, ovšem poskytuje také API pro integraci metrik v jazyce C, Pythonu a Perlu. [1] PCP je používán firmami, jako jsou Red Hat, Netflix, Fujitsu, IBM, bankami a telekomunikačními společnostmi. [2] PCP podporuje rozsáhlé množství platforem. Umožňuje nejen inspekci aktuálních metrik, ale také záznam metrik do logů, s jejichž pomocí lze s metrikami dále pracovat. Jeho distribuovaná architektura umožňuje kolekci metrik z mnoha systémů současně. [1] PCP lze rozdělit do dvou skupin nástrojů:

Monitorující nástroje
: Například utility `pmval` a `pminfo`, které umožní uživateli dotazovat se na data, případně s nimi pracovat jednoduchým způsobem. Dále také PCP nabízí REST API, které umožňuje integrace s nástroji, jako je Grafana. [1]

Nástroje poskytující data
: Programy, které sbírají data z různých domén a zpřístupňují je monitorujícím nástrojům, které nemusejí mít grafické rozhraní pro uživatele. Tyto nástroje se označují jako agenti.

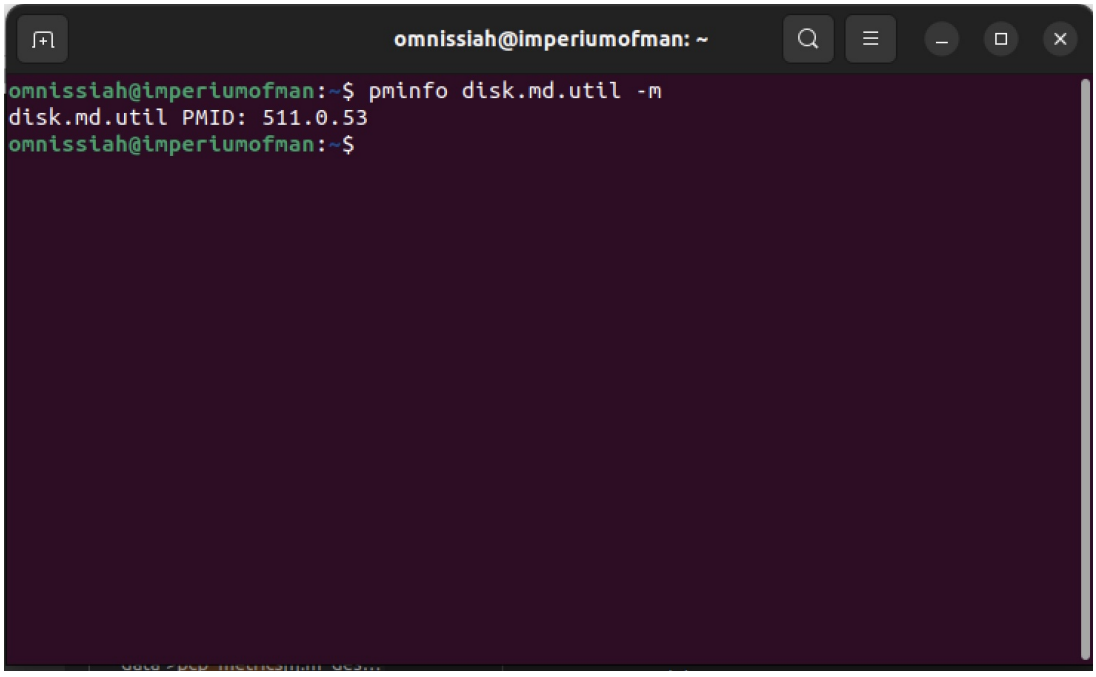


Obrázek 1: Ukázka integrace PCP s Grafanou

[1] Integrace skrze Grafana plugin https://github.com/performancecopilot/grafana-pcp.

1.1 Metriky

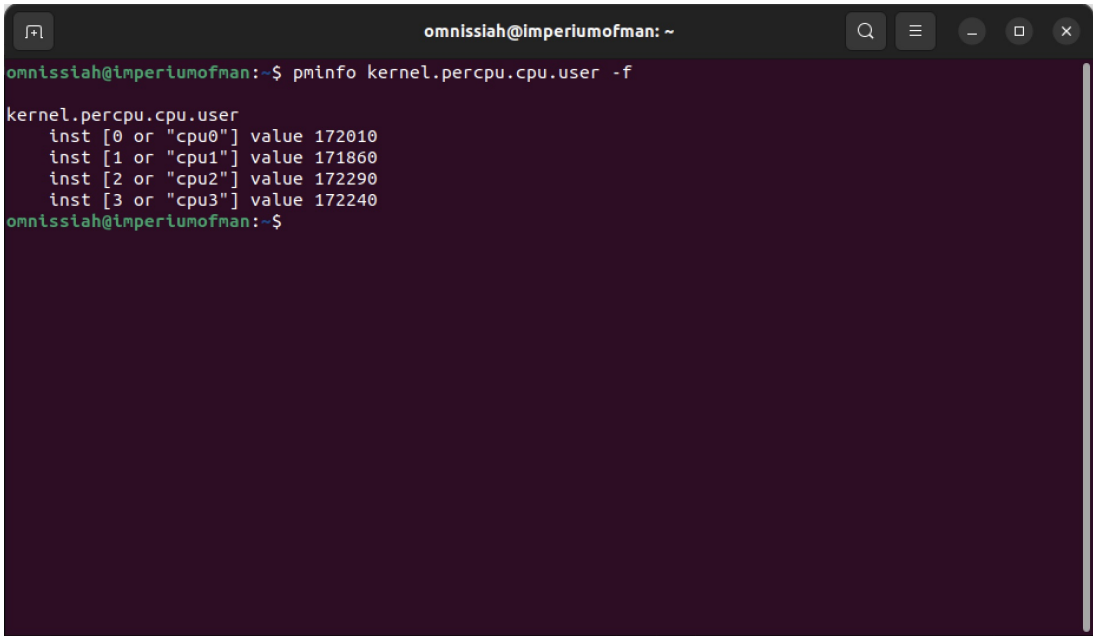
Každá metrika má unikátní identifikátor nebo-li PMID, který se skládá z domény, clusteru a indexu, přičemž každý údaj je kladné celé číslo. Doména metriky je dána agentem, který ji dostane jako parametr při spuštění skrze vlaječky `-d`, zatímco přiřazení clusteru a indexu je již volbou agenta. PMID se zapisuje ve formě *doména.cluster.index*. S přispěním nástroje `pminfo` lze vypsat PMID libovolné metriky pomocí vlaječky `-m`. Kromě PMID má metrika také pro uživatele přívětivější jméno, které je rovněž unikátní napříč metrikami a je mapováno 1 : 1 k identifikátoru metriky. Jméno metriky se zapisuje podobně jako PMID, a to ve tvaru *jmenocast1.jmenocast2.jmenocast3*. Na rozdíl od PMID se může jméno skládat z více než tří částí. Při práci s nástrojem `pminfo` se lze na metriku dotazovat jak pomocí PMID, tak i jejího jména.



Obrázek 2: Vypsání PMID metriky *disk.md.util*

Metriky mohou mít jednu či více hodnot, ovšem datový typ je shodný pro všechny hodnoty dané metriky. [2] Hodnoty metriky v daném okamžiku můžeme vypsat opět pomocí nástroje `pminfo`, a to použitím vlaječky `-f`. Například vícehodnotová metrika *kernel.percpu.cpu.user*, reprezentující čas procesoru strávený exekucí uživatelských programů, má jednu hodnotu pro každé jádro procesoru hosta. [3]

[2] PCP podporuje řadu typů pro hodnoty metriky: double, u64, i64, u32, i32, string apod.

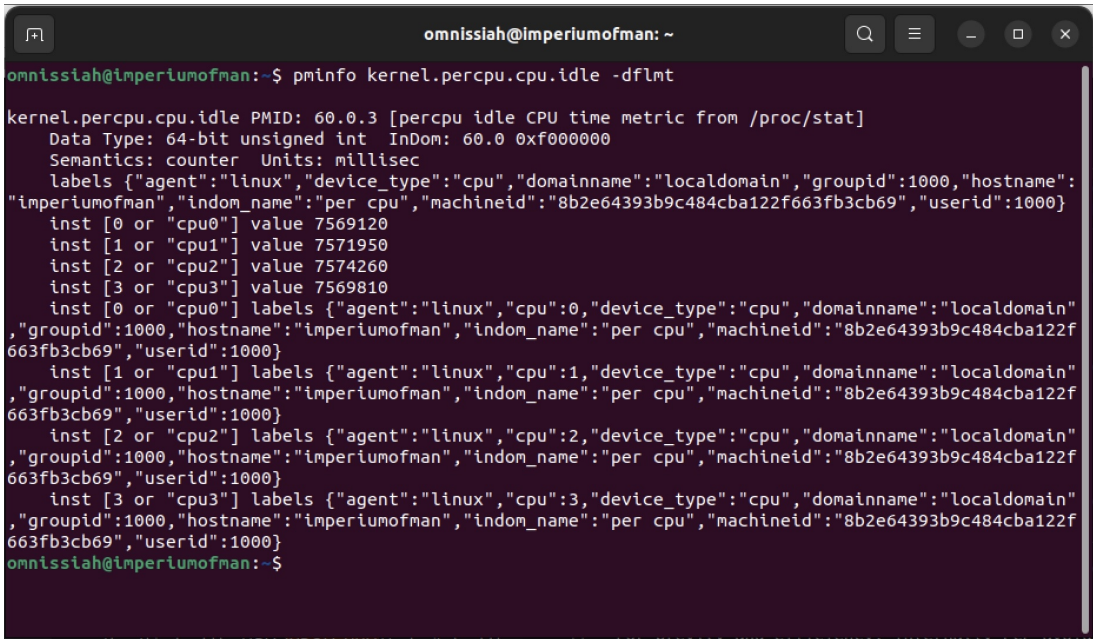


Obrázek 3: Vypsání hodnot metriky *kernel.percpu.cpu.user*

Pojem *hodnota* metriky je spojen s konceptem *instancí* metriky. Má-li metrika jednu hodnotu, má potom jednu instanci. Má-li metrika hodnot více, má i instancí více. Množina instancí metriky se nazývá *doména instancí*. S výjimkou výchozí domény instancí pro metriky s pouze jedinou hodnotou je třeba při definici domény instancí specifikovat dodatečná metadata, umožňující jak PCP (číselný identifikátor), tak i samotnému uživateli (řetězec) rozpoznat hodnoty dané metriky mezi sebou. [3]

Mimo výše uvedená metadata umožňuje PCP také specifikovat na úrovni metriky sémantiku hodnot (např. monotónní čítač a vzorek), jednotku hodnot (např. časové a paměťové jednotky), pomocný text (jednořádkový a víceřádkový), popisky hodnot a další. [3] [3]

[3]Pro uživatele jsou všechna tato metadata dostupná k inspekci použitím různých vlaječek `pminfo`.



Obrázek 4: Vypsání hodnot a metadat metriky *kernel.percpu.cpu.idle*

1.2 Performance Metric Domain Agent

Performance Metric Domain Agent (dále jen 'agent') je program, jehož účelem je kolekce metrik z domény zájmu a jejich expozice monitorujícím nástrojům. PCP nabízí širokou škálu agentů, například: pmdaapache (analýza běhu HTTP serveru Apache), pmdamysql (analýza běhu MySQL databáze), pmdasystemd (analýza SystemD žurnálu) a další. [4] [1] Uživatelé mohou také vytvářet své vlastní agenty. [4].

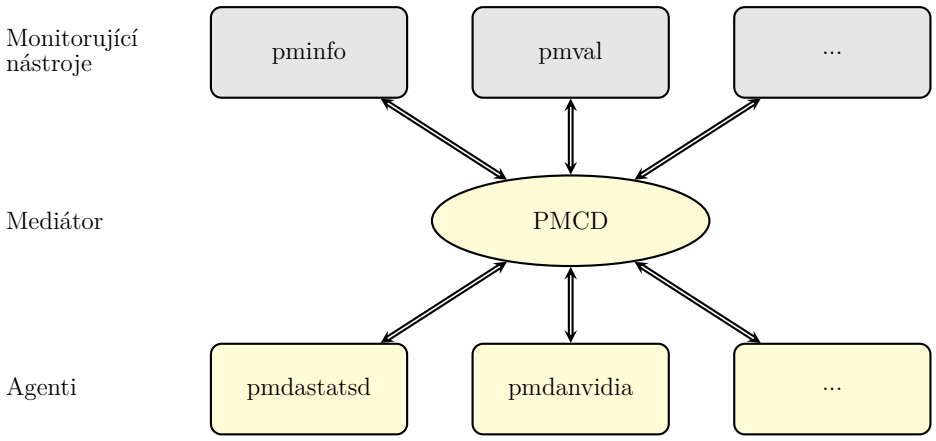
Každý agent má přiřazen unikátní identifikátor označující *doménu* metriky, jež spravuje. Agent tedy nemůže být spuštěn v rámci hosta více než jednou, aniž by došlo ke kolizi těchto domén. [4] [5]

1.3 Performance Metric Collector Daemon

Performance Metric Collector Daemon (dále jen 'PMCD') je démon běžící na každém hostu s PCP, který zastává roli prostředníka koordinujícího komunikaci mezi monitorujícími nástroji, které chtějí obdržet data, a jednotlivými agenty poskytujícími data. Tato komunikace probíhá prostřednictvím API, které PCP poskytuje (dále jen 'PMAPI'). PMCD má odpovědnost také za orchestraci agentů,

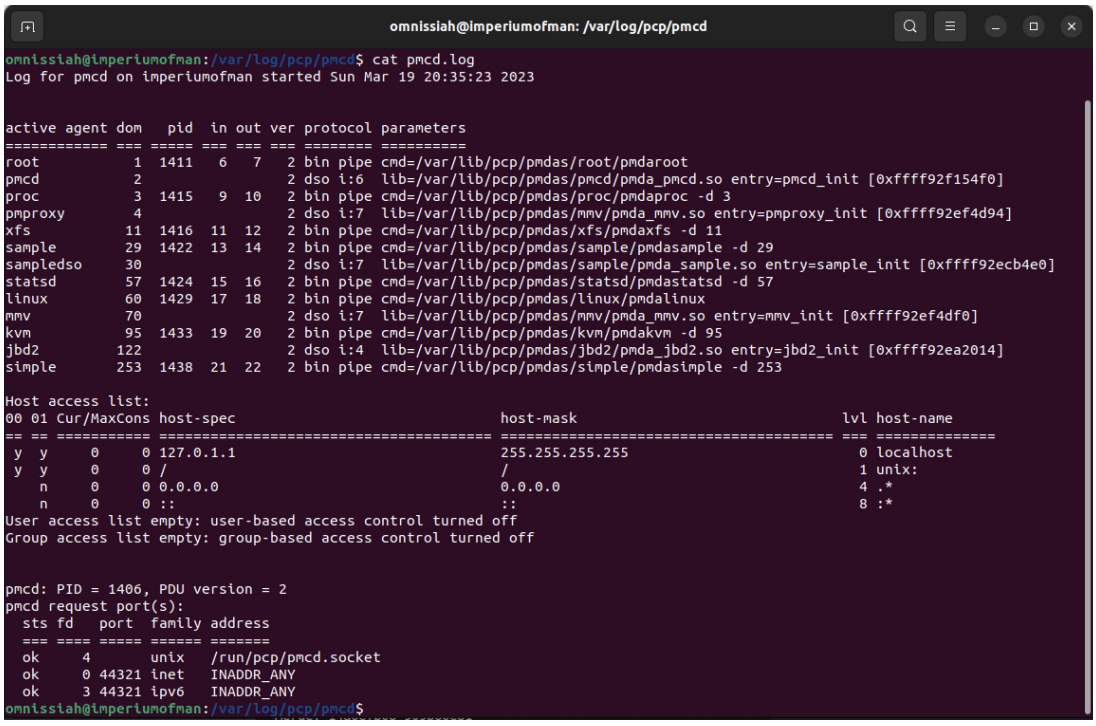
[4] Všechny oficiální agenty lze najít ve složce src/pmdas repozitáře PCP.

[5] Doménový identifikátor lze předat agentovi manuálně přes parametry příkazové řádky. [5] Orchestrace agenta není ovšem odpovědností uživatele, tudíž tato situace není pravděpodobná.



Obrázek 5: Zjednodušený diagram architektury PCP [7]

kteří mohou být spuštěni buďto jako součást procesu PMCD, či jako separátní procesy. [6] [6]



Obrázek 6: Log PMCD obsahující mimo jiné i seznam běžících agentů

[6] V závislosti na tom, zda agent podporuje načtení jako dynamicky sdílený objekt, či běh ve formě démona.

2 StatsD

StatsD je síťový démon, který běží na platformě Node.js. [8] StatsD lze také použít pro označení formátu zpráv reprezentujících metriky, které je démon schopen zpracovat, nebo-li protokol. StatsD je velmi populární. [7] Metriky, které zpracuje, umožňuje dále publikovat do rozsáhlého množství služeb či aplikací, jako jsou například MySQL (relační databáze, integrace možná prostřednictvím fradinni/nodejs-statsd-mysql-back), CouchDB (dokumentová databáze, integrace možná prostřednictvím sysadminmike/couch-statsd-backend) či standardní výstup (integrace prostřednictvím Console backendu, který je součástí StatsD démona). Konzumenty StatsD metrik můžeme také nazývat *backendy*. [9]

Při běhu démon čeká na data (dále metriky) zaslaná přes síťový protokol TCP či UDP, ta zpracuje a výsledek pošle na backend v konfigurovatelném intervalu. Zpracování metriky se liší v závislosti na jejím typu. Metriky obecně dodržují následující formát:

```
<nazev-metriky>:<hodnota>|<typ>
```

Mezi podporované typy metrik patří *counter*, *gauge*, či *duration*. Doména validních hodnot je dána typem metriky, například metriky typu *counter* [8] své hodnoty (celá čísla) v čase sčítají. Backend tedy v tomto případě obdrží již akumulovanou hodnotu. [8]

StatsD metriky jsou organizovány hierarchicky, kdy znak "." ve jméně značí úroveň hierarchii. [9] [8]

3 HDR Histogram

Histogram s podporou pro záznam a analýzu vzorků dat a jejich počtu s konfigurovatelným celočíselným rozsahem s konfigurovatelnou přesností. Přesnost histogramu je vyjádřena počtem platných číslic zaznamenaných hodnot a poskytuje kontrolu nad kvantizací hodnot napříč specifikovaným rozsahem. [10]

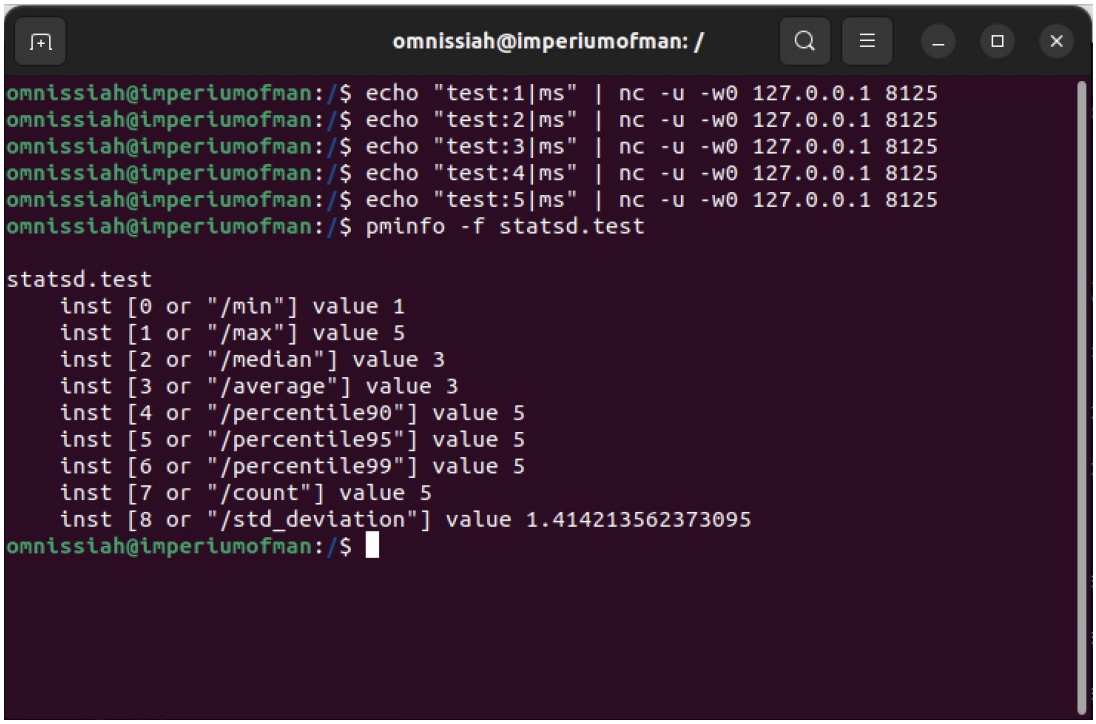
HDR Histogram je navržen pro účel zaznamenání histogramů naměřených hodnot v aplikacích, které jsou citlivé na latenci a výkon. HDR Histogram udržuje konstantní paměťovou a časovou složitost, která přímo závisí na konfiguraci, během zaznamenávání hodnot neprobíhají žádné alokace a práce při zpracování hodnot je konstantní. [10]

Původní implementace HDR Histogramu je napsaná v Javě, je však k dispozici mnoho portů do jiných programovacích jazyků. [10]

V této práci je HDR Histogram využit ke zpracování metrik typu *duration*

[7] Jako metriku popularity můžeme uvažovat počet hvězdiček GitHub repozitáře [8]
[8] Monotónní čítač.
[9] To se dobře mapuje na jmenný prostor PCP metrik, které se organizují podobně.

během agregace zpráv.



Obrázek 7: Ukázka zpracování *duration* metriky HDR Histogramem

4 pmdastatsd

Agent implementovaný v rámci této práce. Jedná se o implementaci podmnožiny funkcionality StatsD démona s podporou jediného backendu, kterým je PCP. [10] Doména dat agenta je dána příchozími StatsD metrikami a množinou předem definovaných metrik, které budou exportovat informace o agentu samém, jako například počet zpracovaných StatsD metrik, počet zahozených StatsD metrik apod. Podporované typy metrik jsou *counter*, *gauge* a *duration*. Metriky typu *duration* mohou být agregovány pomocí HDR Histogramu. Agent také podporuje rozšíření StatsD protokolu, který umožňuje tagovat hodnoty metrik. Tagované metriky dodržují následující formát:

<nazev−metriky>:<hodnota>|<typ>|#<tag1−klic>:<tag1−hodnota>
NEBO
<nazev−metriky>,<tag1−klic>=<tag1−hodnota>:<hodnota>|<typ>

[10] PCP samotný nabízí nemálo nástrojů pro práci se svými metrikami, tudíž díky této tranzitivní relaci vlastně integrace nejen ztratíme, ale i získáme.

V obou případech lze mít tagů více, stačí je oddělit čárkou. Takto tagované hodnoty jsou mapovány do instancí PCP metrik, kdy jedna kombinace tagů odpovídá jedné instanci v případě StatsD metrik typu *counter* a *gauge*, či více instancím v případě typu *duration*. StatsD metriky se stejným názvem, ale rozdílnými tagy jsou v rámci PCP reprezentovány jednou metrikou s více instancemi. Jednotlivé instance jsou pojmenovány dle daných tagů. Agent je implementován v jazyce C.

Hlavní přidanou hodnotou pmdastatsd je v porovnání s již existujícími agenty možnost registrovat nové metriky za chodu agenta a následně je i monitorovat. [11] Ostatní agenti buďto mají doménu metrik předem danou, nebo umožňují uživateli pouze předem specifikovat doménu statickou, která se již za chodu nemůže měnit. [11] Díky tomuto je pmdastatsd ideální pro použití v rámci jiných aplikací, které již nemusejí své metriky v rámci PCP spravovat samy, opět prostřednictvím dalšího agenta, který by byl specifický pro danou aplikaci, ale můžou využít pmdastatsd jako abstrakci nad PCP. Omezení se zde klade na protokol, kterým aplikace s agentem mohou komunikovat, zde tedy StatsD.

4.1 Kód projektu

Vývoj zpočátku probíhal v repozitáři autora. Dnes je tento repozitář již archivován, ovšem je stále dostupný na Githubu Erbenos/pmdastatsd. Jakmile byl agent v provozuschopném stavu, byl proveden merge do PCP repozitáře performancecopilot/pcp a další vývoj probíhal tam. Udržovatelé a vývojáři PCP vzhledem k tomu, že se agent stal de facto součástí PCP, také v kódu agenta (případně jeho testech) prováděli změny v menším rozsahu, například jako reakci na změny v podporovaných platformách [12] oprava drobných chyb [13] či za účelem zohlednění změn v sociálním klimatu a s tím souvisejících hnutích, [14]. Pro hrubý přehled všech kontribucí autora této diplomové práce si lze vyfiltrovat všechny commity GitHub účtu Erbenos před létem 2020. [15]

Vzhledem k mergovací strategii repozitáře PCP jsou v PCP repozitáři také zahrnuty commity dřívějšího repozitáře autora. Ve skutečnosti je v PCP repozitáři více autorových comittů, ale buď se nejedná o zásadní změny v chování

[11] Výjimkou je agent pcp-mmvstatsd, což je dřívější implementace agenta zpracovávající metriky skrze StatsD, který má být nahrazen právě pmdastatsd.

[12] Například commit 922930a GitHub repozitáře performancecopilot/pcp, adresující nekompatibilitu s vývojovou větví distribuce Fedora.

[13] Například commit 6902959 GitHub repozitáře performancecopilot/pcp, adresující chybu při práci s mutexy během výpisu metrik za účelem ladění.

[14] Konkrétně hnutí Woke. Relevantní commit cde7aff GitHub repozitáře performancecopilot/pcp.

[15] Implementace agenta proběhla z podstatné míry během léta 2019 jako součást akce Google Summer of Code 2019. Autor práce se ovšem účastnil s organizací PCP také na Google Summer of Code 2020 a napříč dalšími lety došle k menší míře kontribucí. Filtr zohledňuje tedy commity před touto druhou účastí.

agenta, či jsou pro agenta nerelevantní. Repozitář PCP je také naklonován na médium přiložené k této práci.

V rámci kapitoly pmdastatsd jsou všechny cesty relevantní k lokalitě zdrojů agenta v repozitáři PCP, není-li uvedeno jinak. Bude-li třeba referovat k souborům mimo tuto destinaci, bude cesta obsahovat prefix daný proměnnou. Mapování těchto proměnných je dáno následovně:

$PCP_ROOT
: Cesta k repozitáři PCP.

Zdroje agenta se nachází na cestě $PCP_ROOT/src/pmdas/stats/src .

4.2 Závislosti

Agent využívá kód třetích stran.

4.2.1 chan

Knihovna implementující kanály jazyka Go v jazyce C. [12] Jedná se o jednoduchou abstrakci pro zasílání a přijímání zpráv napříč vlákny. Zasílání může být blokující i neblokující. [12] S pomocí této abstrakce je v agentovi snadno realizována jednosměrná komunikace mezi jednotlivými vlákny.

4.2.2 HdrHistogram_C

Port již zmíněného HDR Histogramu do jazyka C.

4.2.3 Ragel

Ragel je generátor parserů. Reprezentuje je jako konečné automaty, které vytvoří na základě doménově specifického jazyka, kterým uživatel definuje jak parser samotný, tak i zpětná volání, jež se v daných přechodech výsledného automatu zavolají. Takto generované automaty a jejich obslužný kód mohou být embedovány do počtu podporovaných jazyků, včetně jazyka C. Generovaný kód nemá žádné závislosti. [13]

Ragel také umožňuje generovat vizualizace automatů ve formátu DOT, se kterým umí pracovat celá řada programů a který lze konvertovat do jiných formátů. [14] Vizualizace automatu této práce vznikly generací automatu či jeho části, následným převodem DOT souboru do vektorového souboru SVG a následně do bitmapy.

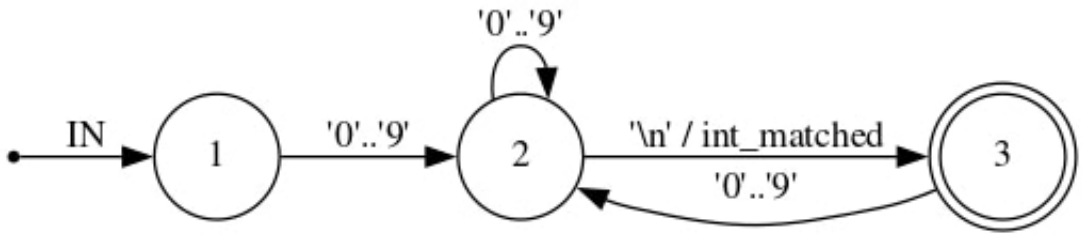
```
#include <stdio.h>
#include <string.h>
// Inicializace proměnných, kód níže je součástí funkce
int cs;
char *p = input;
char *pe = input + strlen(input);
char *eof = pe;
// Blok Ragel kódu
%%{
   # Pojmenuje automat
   machine int_machine;
   # Vypíše statická data stroje... seznam stavů a přechodů
   write data;

   # Registrace callbacku
   action int_matched {
      # kód cílového jazyka, v tomto případě C
      printf("DONE");
   }
   # Stav jeden nebo více znaků
   integer = (
      [0-9]+
   # "\%" callback zavolán při opuštění stavu
   ) \%int_matched;

   # Vstupní bod automatu
   main := ( integer '\n' )+;

   # Vypíše inicializační kód
   write init;
   # Exekuce stroje
   write exec;
}%%
```

Zdrojový kód 1: Automat int_machine parsující celá čísla oddělená řádky



Obrázek 8: Vizualizace ukázkového automatu int_machine

4.2.4 PCP

Komunikace agenta s PMCD je umožněna funkcemi PMAPI, které se nachází v souboru `$PCP_ROOT/src/include/pcp/pmapi.h`. Pro agenta se relevantní podmnožina typů, struktur a unionů, s nimiž část funkcí z PMAPI pracuje, nachází v souboru `$PCP_ROOT/src/include/pcp/pmda.h`.

PCP také zahrnuje knihovny pro práci s hashtabulkami a konfiguračními soubory. Tyto knihovny nejsou vytvořeny autory PCP, nicméně jsou součástí repozitáře kvůli usnadnění distribuce a minimalizace závislostí třetích stran. Jedná se o soubory `$PCP_ROOT/src/external/dict.h` (práce s hashtabulkami) a `$PCP_ROOT/vendor/github.com/benhoyt/inih/ini.h` (knihovna pro práci s .ini soubory).

4.3 Inicializace

Agent může běžet buďto jako démon, či může být načten jako dynamický sdílený objekt (DSO) démonem PMCD. Tento fakt je relevantní pouze pro jeho inicializaci, v ostatních částech programu se forma, kterou agent běží, nezohledňuje. Z hlediska inicializace toto agenta lehce komplikuje, naštěstí PMAPI tento fakt relativně dobře abstrahuje. [4] Toto rozdělení lze vidět na obrázku logů PMCD na stránce 12, sloupec "protocol".

4.3.1 Komunikace s PMCD

Jedním z podstatných rozdílů je způsob, jakým PMCD bude s agentem komunikovat. V obou případech musí agent definovat a namapovat speciální funkce, které budou ošetřovat specifické dotazy PMCD. Je-li agent spuštěn jako démon, jsou tyto dotazy komunikovány prostřednictvím Protocol Data Unit (dále jen 'PDU'). [16] V případě DSO není PDU třeba, protože se agent de facto stane součástí PMCD a výše uvedené funkce mohou potom být volány přímo. [4]

4.3.2 Signály

Běží-li agent jako démon, registruje vlastní procedury pro signály `SIGINT` a `SIGUSR1`. Při obdržení signálu `SIGINT` agent manuálně uvolní všechnu dynamicky alokovanou paměť, což usnadňuje paměťové profilování programu a ladění možných úniků paměti. Signál `SIGUSR1` je registrován za účelem poskytnutí primitivní metody pro ladění programu, přijetí tohoto signálu totiž způsobí vypsání všech sledovaných StatsD metrik pomocí jejich interní reprezentace do souboru. Tato množina metrik se může lišit od množiny metrik, které "vidí" PCP, protože synchronizace těchto dvou množin probíhá líně během komunikace mezi PMCD a agentem, více o ní později.

[16] PDU si může čtenář představit jako strukturu reprezentující jednotlivé dotazy, které PMCD může agentovi klást.

4.3.3 Statické metriky

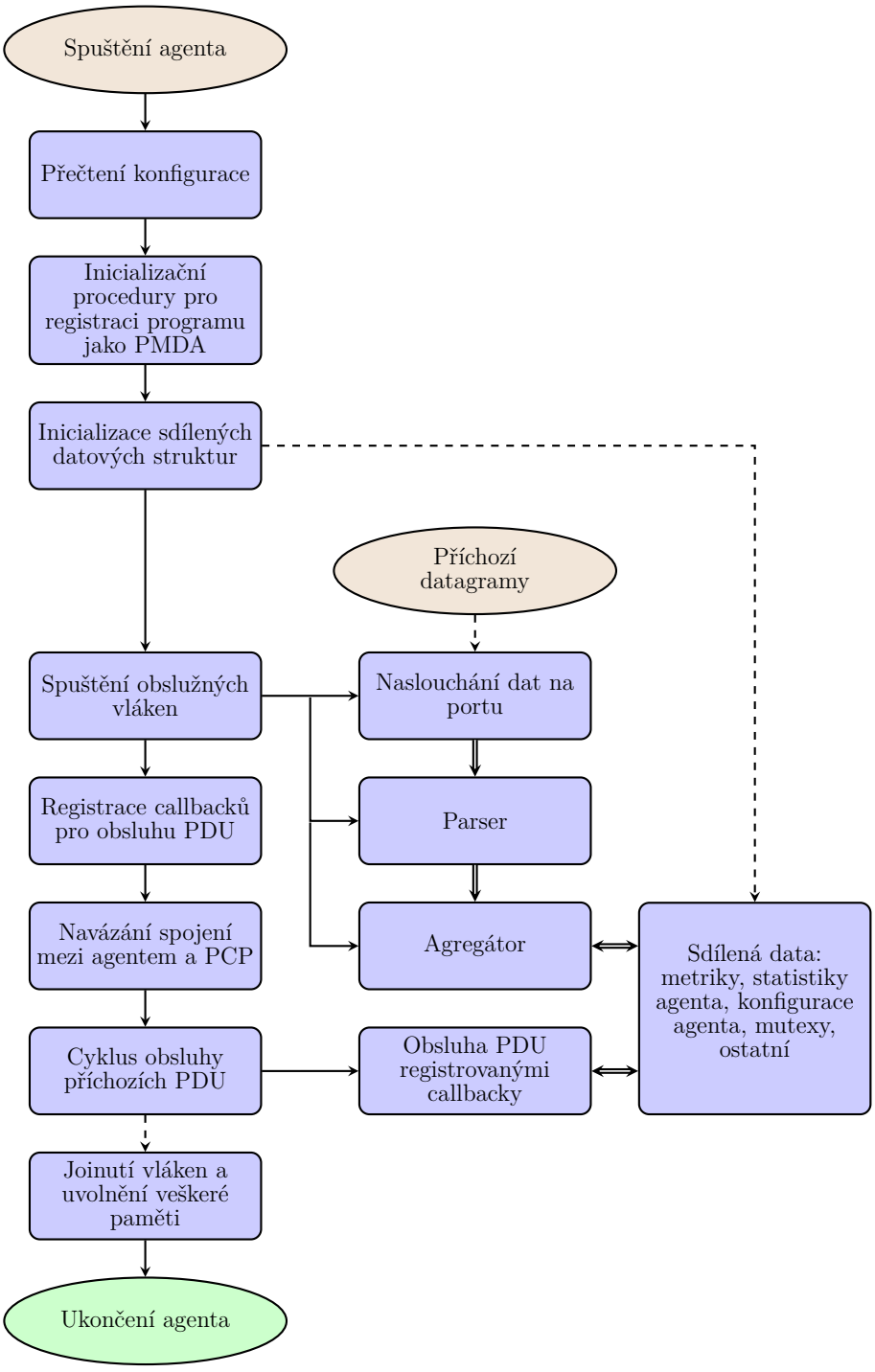
Podmnožina metrik, které bude agent poskytovat, je statická, protože agent předává informace o svém chodu formou metrik. Tyto metriky jsou předem známé a v průběhu času se jejich vlastnosti jako např. sémantika či počet hodnot, které metrika má, nemění, můžeme je tudíž nastavit již při spuštění. Mimo tyto statické metriky a jejich vlastnosti jsou také předem známy jisté vlastnosti StatsD metrik, například počet hodnot, které metrika daného typu má.

4.3.4 Vlákna a jejich komunikace

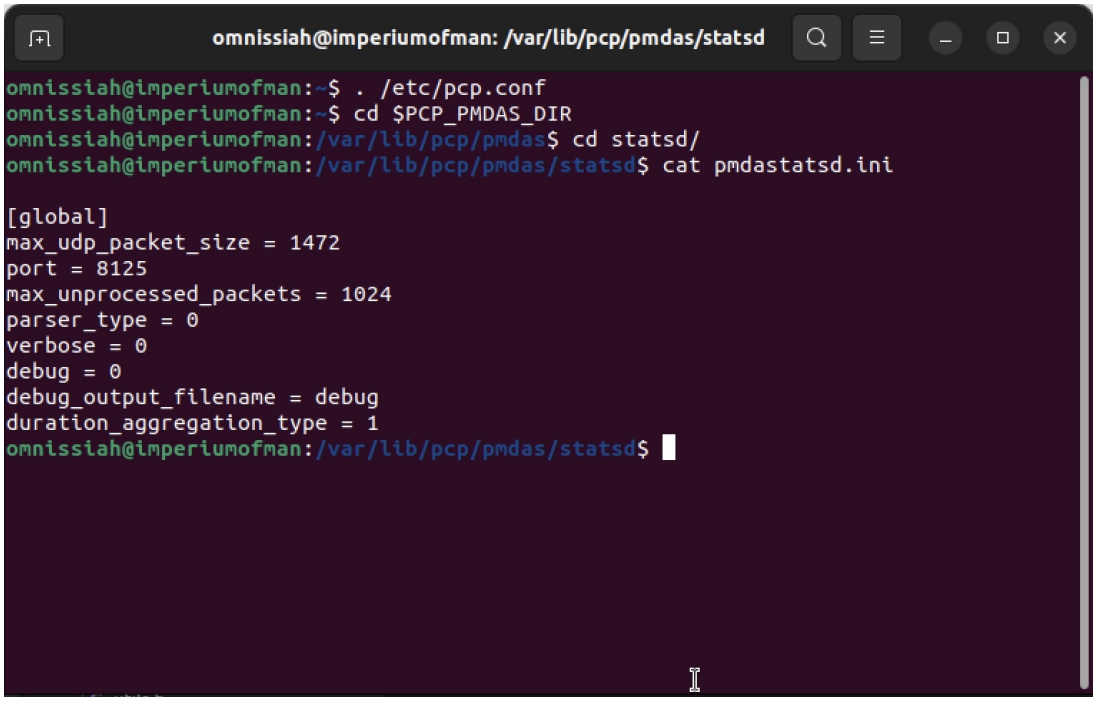
Agent spustí tři vlákna: vlákno, které poslouchá na konfigurovatelném portu a čeká na příchozí datagramy, vlákno parsující obdržené datagramy a vlákno agregující parsované datagramy do interních struktur reprezentujících StatsD metriky. Jednotlivá vlákna mezi sebou komunikují *chan_c* kanály, které mají uživatelem konfigurovatelnou velikost. Kanály existují dva: pro předání příchozích datagramů parseru a pro předání již parsovaných metrik agregátoru. Hlavní program a vlákno agregující naparsované datagramy operují nad stejnými strukturami, které jsou vytvořeny během inicializace.

4.4 Konfigurace

Agentu je předána konfigurace buď pomocí `pmdastatsd.ini` souboru, který se očekává ve stejném adresáři jako spustitelný soubor agenta, nebo pomocí parametrů příkazové řádky. Tyto možnosti konfigurace jsou pokryty souborem `config-reader.c`. Konfigurace pomocí parametrů příkazové řádky mimo hodnoty, které může obsahovat konfigurační soubor, také zahrnuje parametry, jež agentovi předá při spuštění PMCD (viz logy PMCD na stránce 12) a jež slouží spíše než k orchestraci agenta ke konfiguraci jeho chodu, co se práce se StatsD metrikami týče. K práci s těmito extra parametry slouží funkce pmdaGetOptions. Protože PMCD spouští a zastavuje agenta, je pro uživatele praktičtější použít konfigurační soubor.



Obrázek 9: Diagram architektury agenta



Obrázek 10: Ukázka konfiguračního souboru agenta

4.5 Práce se sítí

Datagramy jsou očekávány na portu, který je specifikován v konfiguraci agenta podobně jako maximální velikost příchozího datagramu. Délky nad tento limit jsou zahozeny. Přijde-li datagram, který projde tímto sítem, je jeho hodnota "zapouzdřena" do struktury *unprocessed_statsd_datagram* [17], která je přes kanál poslána ke zpracování parserem.

```
1 // network-listeners.h
2 typedef struct unprocessed_statsd_datagram
3 {
4    char* value;
5 } unprocessed_statsd_datagram;
```

Zdrojový kód 2: Struktura odchozí zprávy síťového vlákna

Při zpracovávání informací z otevřeného socketu se také kontroluje, zda agent obdržel `SIGINT`, nebo zda příchozí datagram obsahuje speciální terminální zprávu.

[17] Vzhledem k tomu, že struktura má pouze jednu hodnotu, technicky vzato by postačil *char **. Vzhledem ke snadnějšímu budoucímu rozšíření jsem volil strukturu.

Pokud ano, je parseru poslána speciální zpráva (opět zapouzdřenáv *unprocessed_statsd_datagram*), která slouží jako terminátor pro ukončení spojení a úklid paměti vlákna.

Výše uvedená logika se nachází v souboru `network-listener.c`.

4.6 Parsování zpráv

Agent obsahuje dva parsery: *BASIC* a *RAGEL*. Který z nich je použit ke zpracování zpráv, závisí na konfiguraci agenta. Z hlediska rozpoznávání zpráv se oba parsery shodují. Proces parsování zpráv je pokrytý souborem `parsers.c`. Jakmile datagram projde parserem, je výsledek operace reprezentován strukturou *parser_to_aggregator_message* a data jsou odeslána přes kanál k agregaci.

```
1 // parsers.h
2 typedef struct parser_to_aggregator_message
3 {
4    struct statsd_datagram* data;
5    enum PARSER_RESULT_TYPE type;
6    unsigned long time;
7 } parser_to_aggregator_message;
```

Zdrojový kód 3: Struktura odchozí zprávy vlákna parseru

Ve struktuře je prozatím již naparsovaná metrika uložena v hodnotě *struct statsd_datagram* data*, která se skládá z:

char* name
: Jméno metriky.

enum METRIC_TYPE type
: Typ metriky. Může být *counter*, *gauge*, či *duration*.

char* tags
: Tagy metriky reprezentované formátem JSON.

int tag_pair_count
: Počet párů v JSONu uloženém v hodnotě **char* tags**.

enum SIGN explicit_sign
: Znaménko hodnoty metriky. Sémantika metriky se může lišit pro hodnoty ""+2"" a ""2"", tudíž je třeba dodatečně rozlišit tyto dva stavy.

double value
: Hodnota metriky bez znaménka.

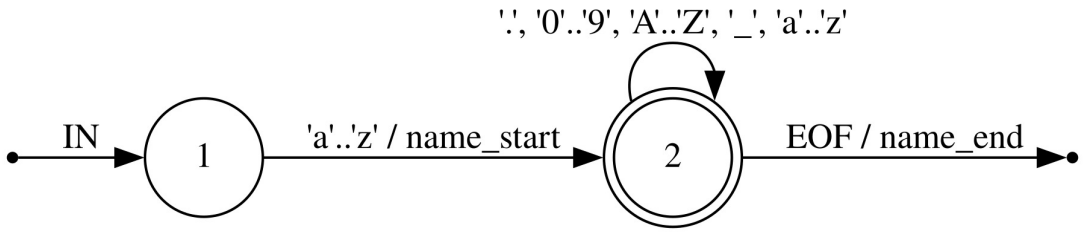
Mezi parsováním jednotlivých datagramů se také zjišťuje, zda agent obdržel `SIGINT`, pokud ano, začne zahazovat příchozí datagramy, které jsou již drženy v bufferu kanálu. Přijde-li zpráva reprezentující terminátor, pošle se agregátoru terminující zpráva a vlákno se ukončí.

```
1  // parsers.h
2  typedef struct statsd_datagram
3  {
4     char* name;
5     enum METRIC_TYPE type;
6     char* tags;
7     int tags_pair_count;
8     enum SIGN explicit_sign;
9     double value;
10 } statsd_datagram;
```

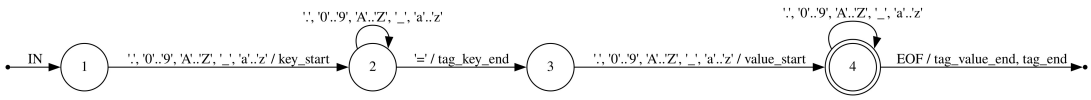
Zdrojový kód 4: Struktura již parsované metriky

4.6.1 Ragel parser

Soubor `parser-ragel.rl` obsahuje zdrojový kód *RAGEL* parseru. Příkazem `ragel -C parsel-ragel.rl` lze expandovat Ragel – specifický kód v daném souboru na kód jazyka C. Agent je distribuován společně přímo s výstupem Ragelu, který je reprezentovaný souborem `parser-ragel.c`. Vizualizace celého automatu je součástí přiloženého média. [18]



Obrázek 11: Vizualizace části parsovacího automatu – jméno metriky



Obrázek 12: Vizualizace části parsovacího automatu – tag před hodnotou

4.6.2 Basic parser

BASIC parser v souboru `parser-basic.c` se od *RAGEL* parseru liší jen tím, že pro jeho implementaci není použit Ragel. *BASIC* parser byl implementován

[18] Výsledný graf je příliš velký pro vložení do jednoho listu A4.

jako první, sloužil hlavně k testování kódu, protože osvojení si Ragelu bylo časově náročné. Nyní je již redundantní a mohl by být odebrán především proto, že *RAGEL* parser je mnohem snazší udržovat. V kódu byl zatím ponechán.

4.7 Agregace zpráv

Agregace zpráv se odvíjí od logiky v souboru `aggregators.c` a zahrnuje jak metriku obsaženou ve zprávě, která přijde kanálem, tak i metriky reprezentující interní statistiky agenta.

4.7.1 Interní statistiky

Nezávisle na tom, zda datagram obsahuje validní [19], či nevalidní řetězec, k agregaci je vždy poslána zpráva ve formě struktury *parser_to_aggregator_message*. Agent totiž neexportuje jenom hodnoty *StatsD* metrik, ale i metriky reflektující chod agenta samotného. Jedná se o údaje jako počet příchozích datagramů, počet zahozených datagramů, počet úspěšně naparsovaných datagramů a tak dále. Všechny hodnoty reflektující tyto interní statistiky jsou drženy v instanci struktury *pmda_stats*.

```
1  // aggregator-stats.h
2  typedef struct pmda_stats {
3     size_t received;
4     size_t parsed;
5     size_t dropped;
6     size_t aggregated;
7     size_t time_spent_parsing;
8     size_t time_spent_aggregating;
9     struct metric_counters* metrics_recorded;
10 } pmda_stats;
```

Zdrojový kód 5: Reprezentace statistik agenta

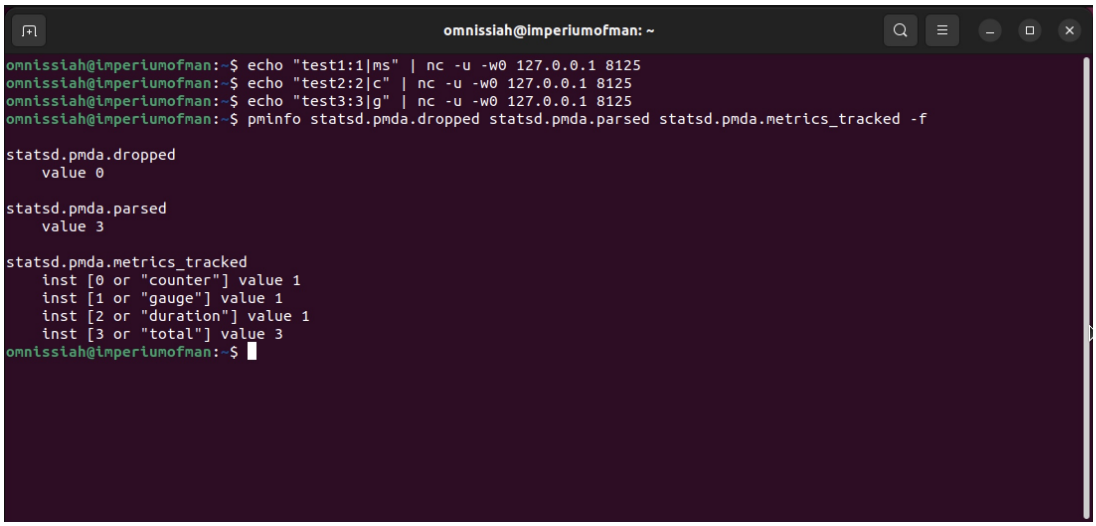
Protože agregující vlákno i obsluha zpětných volání PCP manipulují s těmito daty, je nutné tyto operace synchronizovat. Z toho důvodu se přistupuje ke struktuře *pmda_stats* pouze skrze *pmda_stats_container*, což je struktura disponující mutexem.

Po příjmu zprávy parseru se nejprve zkontroluje, zda parsování proběhlo úspěšně, a pokud ne [20], rovnou je tato skutečnost zaznamenána do interních statistik. V opačném případě se pokračuje agregací metriky do interních struktur. V závislosti na výsledku tohoto procesu se opět aktualizují interní statistiky.

[19] takový řetězec, jehož průchod parsujícím automatem skončí v konečném stavu automatu.
[20], ke zjištění tohoto faktu slouží hodnota *parser_to_aggregator_message.type*.

```
// aggregator-stats.h
// ..
typedef struct pmda_stats_container {
   struct pmda_stats* stats;
   pthread_mutex_t mutex;
} pmda_stats_container;
```

Zdrojový kód 6: Reprezentace kontejneru statistik agenta



Obrázek 13: Ukázka interních statistik

4.7.2 StatsD metriky

Pokud byl obsah datagramu parsován úspěšně, je daná metrika agregována pomocí funkce *process_metric* ze souboru `aggregator_metrics.c`.

Podobně jako pro interní statistiky je přístup k StatsD metrikám umožněn skrze kontejner *pmda_metrics_container*, který je opatřen mutexem.

```
// aggregator-metrics.h
typedef struct pmda_metrics_container {
   metrics* metrics;
   struct pmda_metrics_dict_privdata* metrics_privdata;
   size_t generation;
   pthread_mutex_t mutex;
} pmda_metrics_container;
```

Zdrojový kód 7: Reprezentace kontejneru StatsD metrik

Metriky jsou uloženy do hashmapy *metrics* metrics*. Klíčem záznamu v ha-

shmapě je název dané metriky. [21]

Kvůli existenci statických metrik by mohlo dojít ke kolizím jmen se StatsD metrikami. Pokud například agent o sobě exportuje metriku "pmda.received", reprezentující počet přijatých datagramů, nedávalo by smysl také umožnit přijetí a následné zpracování datagramu s hodnotou "pmda.received:4|c". Proto jsou jistá jména vymezena pro použití statickými metrikami. Během agregace se datagramy s kolidujícími metrikami zahazují. Uživatelům je doporučeno nepoužívat celý jmenný prostor daný prefixem "pmda".

Opomineme-li zatím práci s otagovanými metrikami, je agregace metrik jednoduchý proces. Agent zkontroluje, zda již pro onu metriku existuje v hashmapě záznam, a pokud ano, tak je aktualizuje, pokud ne, tak ji vytvoří, v obou případech se ale synchronizuje mutexem.

Operace manipulující s metrikami jsou strukturovány podobně, jako by tomu bylo v objektově orientovaném programování, kdy by byla metrika reprezentována třídou a jednotlivé operace metodami rozhraní, které by daná metrika implementovala. V jazyku C podpora pro třídy samozřejmě není. [22] V kódu je implementace rozhraní pomyslných tříd nahrazena jednotlivými soubory, pro každý typ metriky zvlášť.

Každý typ metriky musí pro její agregaci implementovat následující operace:

Vytvořit hodnotu
: Alokace paměti, nastavení první hodnoty.

Upravit hodnotu
: Změna hodnoty již vytvořeného záznamu, případná realokace paměti.

Uvolnit hodnotu
: Uvolnění paměti.

Vypsat hodnotu
: Vypsání hodnoty za účelem ladění. Volá se jako reakce na příchozí signál `SIGUSR1`.

Pro každý typ metriky jsou tyto funkce definované v souborech zvlášť:

`aggregator-metric-counter.c`
: Manipulace s hodnotou metriky *counter*.

`aggregator-metric-gauge.c`
: Manipulace s hodnotou metriky *gauge*.

[21] Tedy pro metriku "user_loggedin:1|c" by byl klíč "user_loggedin".

[22] To ovšem neznamená, že by programátor nemohl implementovat systém umožňující objektově orientované programování v jazyce C.

`aggregator-metric-duration.c`
: Manipulace s hodnotou metriky *duration*.
Logika se dále dělí na `aggregator-metric-duration-exact.c` a `aggregator-metric-duration-hdr.c`, a to dle konfigurace agenta pro strategii agregace metrik typu *duration*.

```
1  // aggregator-metrics.h
2  /**
3   * Frees gauge metric value
4   * @arg config
5   * @arg value - value to be freed
6   */
7  void
8  free_gauge_value(struct agent_config* config, void* value) {
9     (void)config;
10    if (value != NULL) {
11       free(value);
12    }
13 }
```

Zdrojový kód 8: Uvolnění hodnoty metriky typu gauge

Během agregace se tedy dynamicky vybírá, které z výše uvedených funkcí se mají zavolat v závislosti na typu právě agregované metriky. Všechny implementace těchto funkcí mají stejnou signaturu.

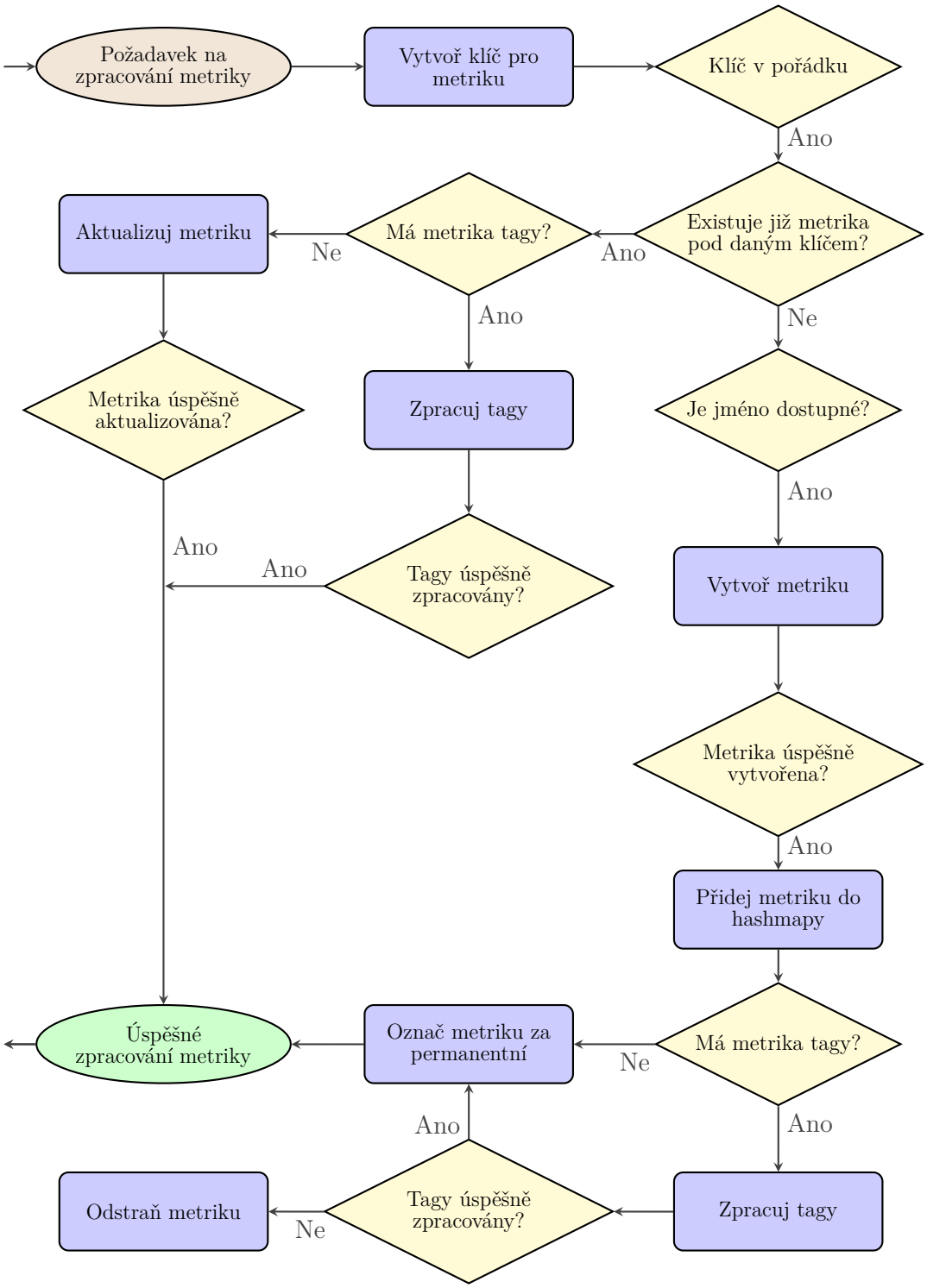
Metrika je v paměti reprezentována strukturou *metric*, která se skládá z:

char* name
: Jméno metriky. Zároveň slouží jako klíč do hashmapy.

int pernament
: Vlaječka, jež určuje, zda je metrika stabilní. Vytvoření metriky a vytvoření otagované hodnoty metriky jsou dvě operace, které jsou synchronizovány zvlášť. Například v okamžiku, kdy dorazí datagram "user_loggout,id=1:1|c", vytvoří se nejprve záznam o metrice a až posléze záznam k popisku "id ". Je totiž žádoucí, aby PMCD nevědělo o nestabilních metrikách, je třeba tyto dva stavy rozlišit.

struct metric_metadata* meta
: Metadata metriky, která jsou relevantní z hlediska komunikace s PMCD, a promítnutí interní reprezentace StatsD metrik do reprezentace metrik, které PCP rozumí. Při agregaci se pouze inicializuje.

labels* children
: Hashmapa pro otagované hodnoty dané metriky. Klíčem záznamu je JSON reprezentace tagů.



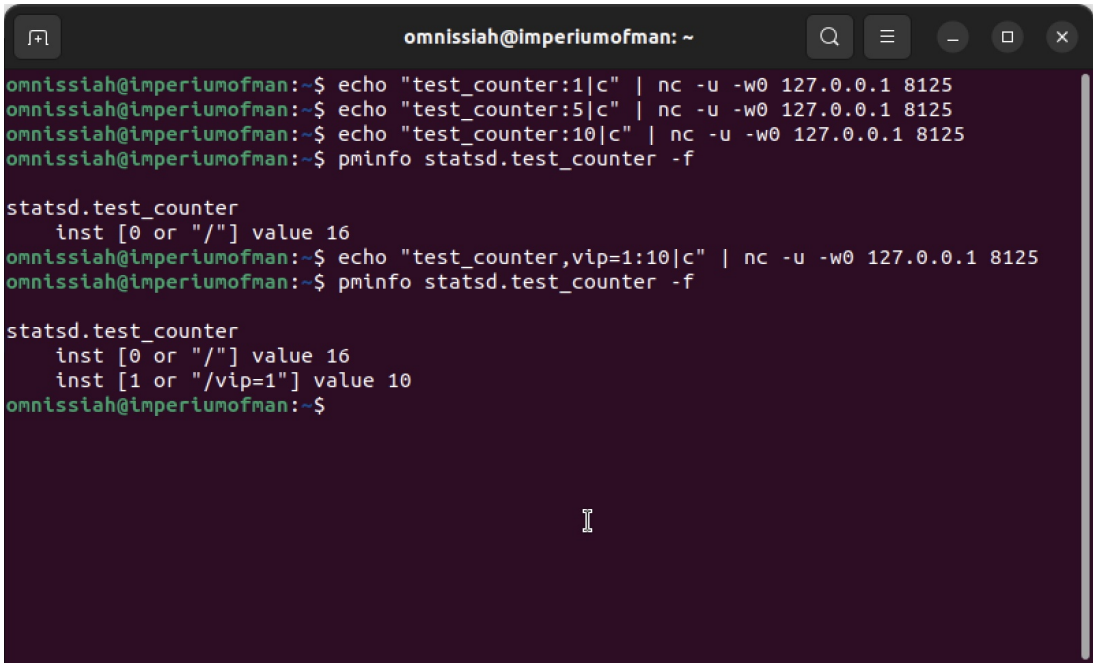
Obrázek 14: Diagram algoritmu zpracování datagramu metriky

enum METRIC_TYPE type
: Typ metriky.

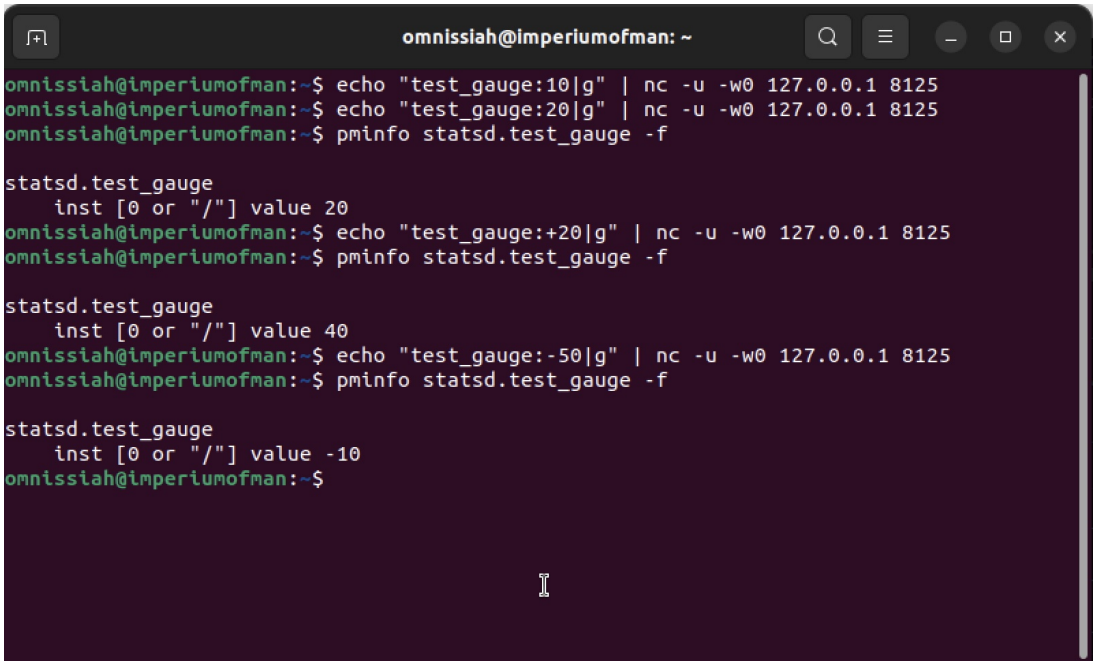
void* value
: Hodnota dané metriky. NULL, pokud metrika nemá hodnotu bez tagů.

```
1 // aggregator-metrics.h
2 typedef struct metric {
3    char* name;
4    int pernament;
5    struct metric_metadata* meta;
6    labels* children;
7    enum METRIC_TYPE type;
8    void* value;
9 } metric;
```

Zdrojový kód 9: Reprezentace StatsD metriky



Obrázek 15: Ukázka zpracování *counter* metrik



Obrázek 16: Ukázka zpracování *gauge* metrik

Otagované hodnoty metrik jsou uloženy v další hashmapě na úrovni dané metriky ve struktuře *metric* v *labels* children*. Klíčem otagovaného záznamu v hashmapě je *JSON* reprezentace tagů přítomných v procesovaném datagramu.

Otagovaná metrika je společně s pomocnými metadaty uložena ve struktuře *metric_label*, která se skládá z:

char* labels
: JSON reprezentace naparsovaných tagů datagramu převzatá z

int pair_count
: Počet jednotlivých tagů záznamu.

struct metric_label_metadata* meta
: Struktura s pomocnými metadaty pro integraci s PCP. Nyní obsah zahrnuje pouze hodnotu **char* instance_label_segment_str**, což je popisek instance, který se zobrazí při výpisu otagované hodnoty monitorujícími nástroji PCP.

enum METRIC_TYPE type
: Typ metriky.

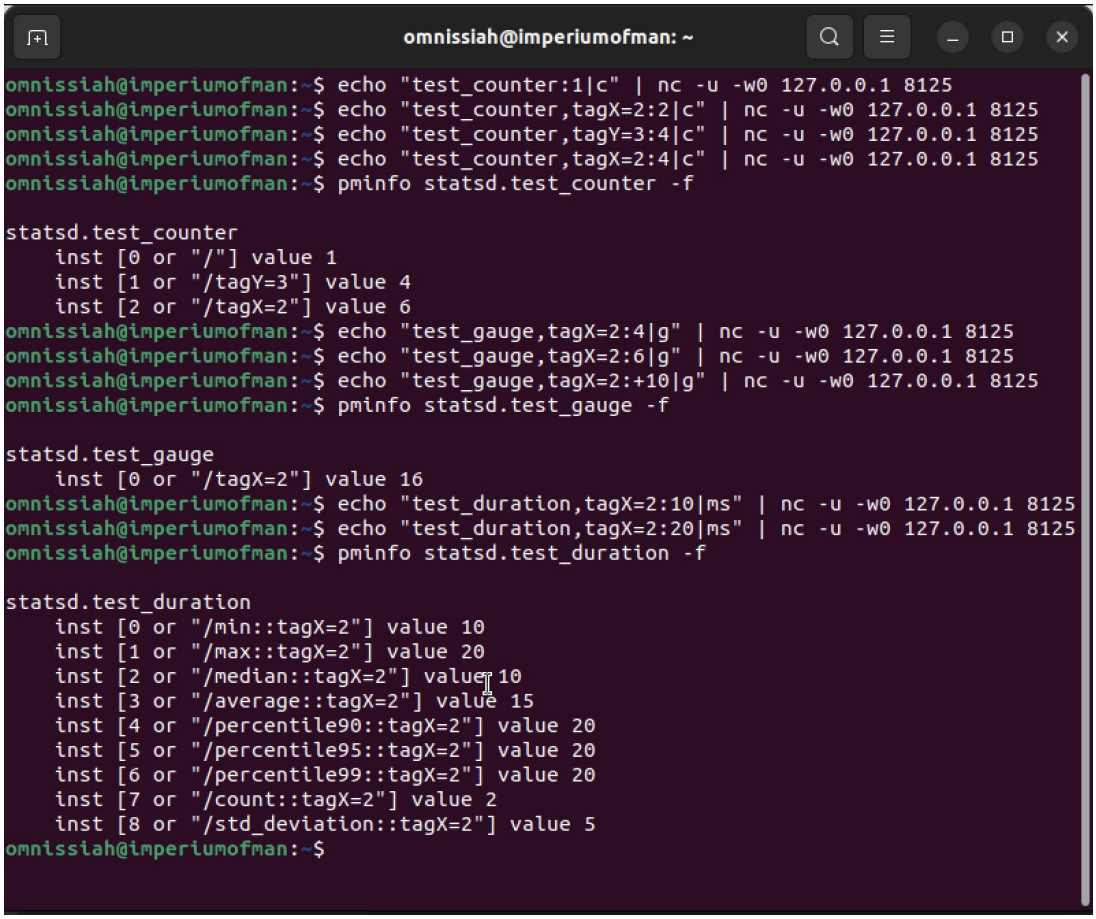
void* value
: Hodnota dané otagované metriky.

Práce s otagovanými hodnotami metrik je pokryta funkcí *process_labelled_datagram* ze souboru `aggregator_metric_labels.c`, která je zavolána v rámci funkce

```
1 // aggregator-metrics.h
2 typedef struct metric_label {
3     char* labels;
4     int pair_count;
5     struct metric_label_metadata* meta;
6     enum METRIC_TYPE type; // either this or parent reference, so
          that we know how to free void* value
7     void* value;
8 } metric_label;
```

Zdrojový kód 10: Reprezentace tagované StatsD metriky

process_metric. Jediný rozdíl mezi neotagovanými a otagovanými hodnotami metrik jsou tagy samotné, tudíž i zde se dynamicky volají výše uvedené funkce, které jsou implementací pomyslného rozhraní.



Obrázek 17: Ukázka zpracování tagovaných metrik

Při přidání, či odebrání metriky, ať už s tagy, či bez nich, je v kontejneru

metrik, který je daný již zmíněnou strukturou *pmda_metrics_container*, inkrementovaná hodnota *generation*. Toto je kvůli nutnosti synchronizovat interní reprezentaci StatsD metrik s reprezentací metrik v kontextu PCP. Tato synchronizace probíhá během obsluhy dotazů PMCD. Během komunikace s PMCD se porovnává *generation* interních metrik a *generation* ve strukturách specifických pro obsluhu této komunikace, reprezentující "generaci".

Nyní zpátky k pomocným metadatům, která jsou držena v hodnotě *struct metric_metadata* meta* struktury *metric*. Ta se skládají z:

char* pcp_name
: Jméno metriky v kontextu PCP. Je nezbytné, protože každý agent dostane přiřazený jmenný prostor. Pro agenta to znamená prefixování všech jmen StatsD metrik řetězcem "statsd.". Přijde-li datagram s "user_loggedin:1|c", monitorujícím nástrojům PCP bude tato metrika prezentována ve formě "statsd.user_loggedin".

struct pmdaInstid_map* pcp_instance_map
: Pomocná struktura držící pole klíčů do hashmapy s otagovanými hodnotami dané metriky.

double sampling
: Nepoužívá se, artifakt vývoje.

pmID pmid
: Bitová mapa sloužící jako unikátní identifikátor dané metriky. Inicializováno na *PM_ID_NULL*, reprezentující invalidní identifikátor. [4]

pmInDom pmindom
: Unikátní identifikátor domény instancí dané metriky. [4]

size_t pcp_instance_domain_index
: Index do pole držícího struktury *pmdaIndom* reprezentující doménu instancí metriky v rámci *PCP*. [4]

site_t pcp_metric_index
: Index do pole držícího struktury reprezentující metriku v rámci *PCP*. Zatímco již zmíněná struktura *metric* reprezentuje StatsD metriku interně v rámci agenta, metriku lze také reprezentovat unifikovaným způsobem při integraci s PCP strukturou *pmdaMetric*. Tato struktura obsahuje metadata jako datový typ hodnoty metriky, identifikátor metriky, sémantiku hodnoty a její dimenzi či jednotku. [4] Relevantní především pro zpětná volání zajišťující integraci s PCP.

int pcp_instance_change_requested
: Vlajka určující, zda je třeba synchronizovat doménu instancí pro danou metriku s interní reprezentací otagovaných hodnot metriky.

Výjimkou *int pcp_instance_change_requested*, která je v rámci agregace průběžně nastavována pokaždé, když se změní množina otagovaných metrik, jsou zbylé hodnoty pouze inicializovány.

```
1  // aggregator-metrics.h
2  typedef struct metric_metadata {
3     char* pcp_name;
4     struct pmdaInstid_map* pcp_instance_map;
5     double sampling;
6     pmID pmid;
7     pmInDom pmindom;
8     size_t pcp_instance_domain_index;
9     size_t pcp_metric_index;
10    int pcp_instance_change_requested;
11 } metric_metadata;
```

Zdrojový kód 11: Struktura pomocných metadat metriky

4.8 Komunikace s PMCD

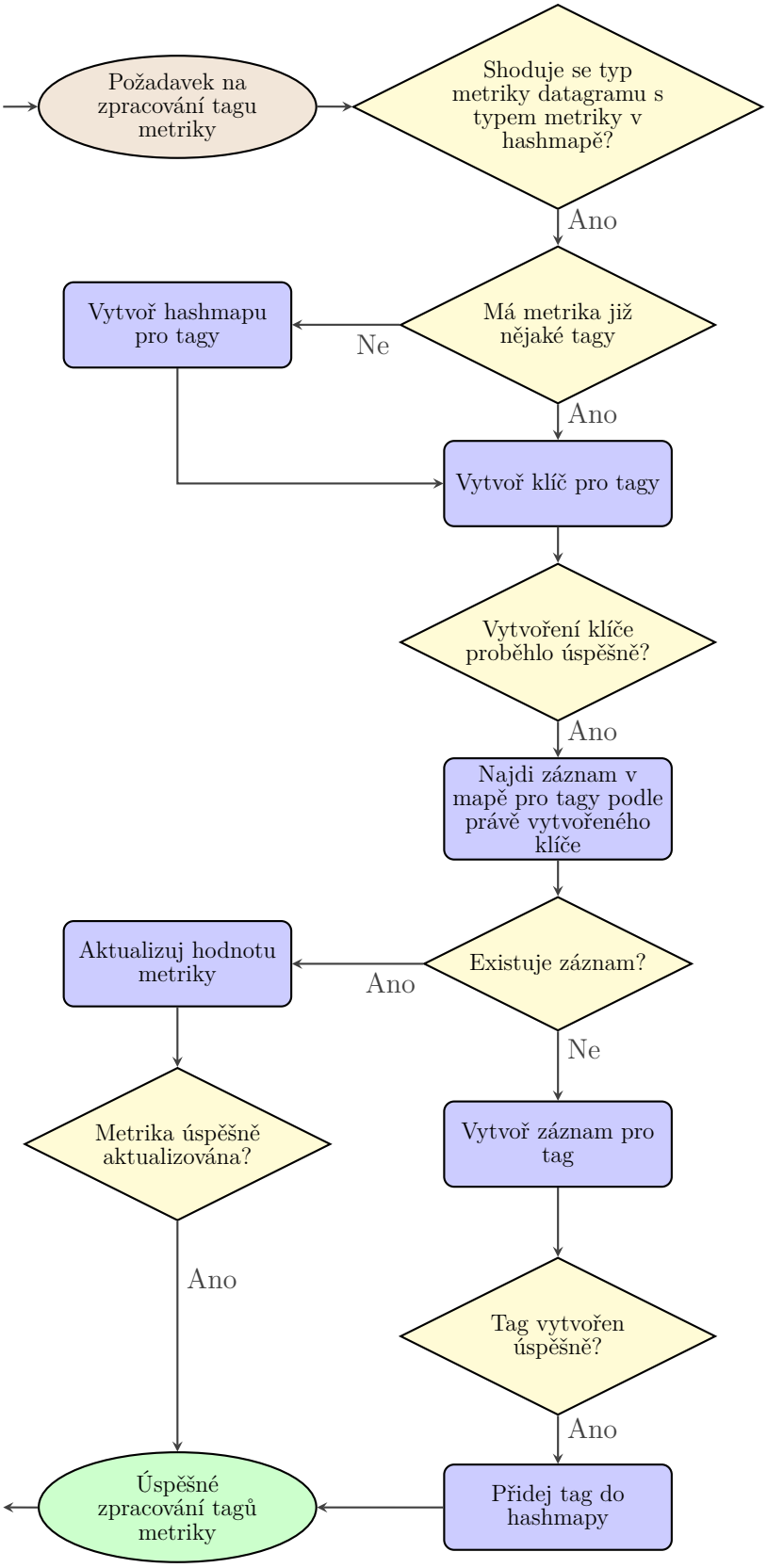
Aby agent mohl prezentovat metriky své domény, obsluhuje požadavky monitorovacích nástrojů, které jsou agentovi komunikovány skrze PMCD, jenž slouží jako mediátor těchto požadavků. Tato obsluha je implementována specifikováním funkcí, jež jsou v závislosti na typu požadavku volány s různými parametry. Současně agenti nemusí implementovat všechny funkce, protože PCP již přichází s výchozí obsluhou, která je pro většinu případů dostačující. [4]

Je třeba odlišit funkce, které se volají za účelem obsluhy požadavků PMCD, od dalších funkcí, jež se registrují pomocí funkcí *pmdaSetFetchCallBack* a *pmdaSetLabelCallBack*, jejichž účelem je definovat způsob, jakým agent předává hodnoty jednotlivých metrik a popisků. Jedná se o další úroveň abstrakce, která usnadňuje implementaci agenta. Jeden požadavek od PMCD totiž může obsahovat žádost o vydání více než jedné metriky. V takovém případě by byla obsluha požadavku zavolána jednou, ale funkce registrovaná přes *pmdaSetFetchCallback* je volána pro každou požadovanou metriku zvlášť. Obdobně pro popisky a *pmdaSetLabelCallback*. Toto platí za předpokladu výchozí obsluhy jednotlivých požadavků. [4] Mapování zmíněných obsluh je nastaveno pomocí struktury *pmdaInterface*. [4]

Funkce obsluhující požadavky PMCD mají jména s prefixem "statsd_" a nachází se v souboru `pmda-callbacks.c`.

Vzhledem k dynamičnosti metrik, které agent může poskytnout [23], je třeba synchronizovat interní reprezentaci StatsD metrik s množinou metrik a jejich instancí, které "vidí" PCP. Za účelem synchronizace tedy agent registruje vlastní

[23] Metriky a instance se totiž v čase mohou měnit dle toho, jak přichází datagramy.



Obrázek 18: Diagram algoritmu zpracování datagramu metriky s tagy

```
// pmdastatsd.c
void
__PMDA_INIT_CALL
statsd_init(pmdaInterface *dispatch) {
  ...
  dispatch->version.seven.fetch = statsd_fetch;
  dispatch->version.seven.desc = statsd_desc;
  dispatch->version.seven.text = statsd_text;
  dispatch->version.seven.instance = statsd_instance;
  dispatch->version.seven.pmid = statsd_pmid;
  dispatch->version.seven.name = statsd_name;
  dispatch->version.seven.children = statsd_children;
  dispatch->version.seven.label = statsd_label;
  pmdaSetFetchCallBack(dispatch, statsd_fetch_callback);
  pmdaSetLabelCallBack(dispatch, statsd_label_callback);
  ...
}
```

Zdrojový kód 12: Registrace vlastní obsluhy PDU zpráv

obsluhu nejen pro poskytnutí hodnot a popisků instancí jednotlivých metrik pomocí *pmdaSetFetchCallback* a *pmdaSetLabelCallback*, ale také vlastní obsluhu požadavků PMCD. Ve většině případů se jedná o jednoduché funkce, které dodatečně provedou zmíněnou synchronizaci a poté předají kontrolu funkcím, jež by sloužily jako výchozí obsluha jednotlivých požadavků. Logika zmíněné synchronizace bude popsána později.

```
// pmda-callbacks.c
int
statsd_instance(pmInDom in_dom, int inst, char* name, pmInResult**
    result, pmdaExt* pmda) {
   statsd_possible_reload(pmda);
   return pmdaInstance(in_dom, inst, name, result, pmda);
}
```

Zdrojový kód 13: Příklad obsluhy PDU, která zajistí synchonizaci interní reprezentace StatsD metrik s PCP reprezentací metrik

Výjimkou je funkce *statsd_text*, která slouží jako obsluha požadavku pro obdržení popisku dané metriky. Výchozí obslužná funkce *pmdaText* totiž hledá tento text ve specificky formátovaném souboru `help` v adresáři agenta. [4] Zatímco by nyní takové řešení bylo technicky dostačující, protože agent nabízí textové popisky pouze pro metriky reprezentující interní statistiky, chtěli bychom v budoucnu přijít se způsobem, který by umožnil uživateli přidat tyto popisky přímo do jednotlivých datagramů dynamických metrik, stejně bychom se museli vzdát toho mít soubor `help`.

Všem obslužným funkcím je předána jako poslední argument struktura *pmdaExt*, která reprezentuje kontext volání funkce a jež kromě držení interních informací umožňuje také nastavit uživatelem definovaná data. [4] Agent uchovává tato data ve struktuře *pmda_data_extension*. Tato struktura se skládá z:

struct agent_config* config
: Konfigurace agenta. Reprezentuje konfiguraci předanou agentovi buď přes příkazovou řádku, či konfiguračním souborem.

struct pmda_metrics_container* metrics_storage
: Úložiště StatsD metrik.

struct pmda_stats_container* stats_storage
: Úložiště interních metrik statistik agenta.

pmdaMetric* pcp_metrics
: Pole PCP metrik.

pmdaIndom* pcp_instance_domains
: Pole PCP doménových instancí.

pmdaNameSpace* pcp_pmns
: Reprezentace jmenného prostoru PCP metrik agenta.

dict* instance_map
: Hashmapa zajišťující mapování z unikátních identifikátorů instancí daných strukturou *pmInDom* (respektive návratovou hodnotou funkce *pmInDomStr*). Obslužné funkce vždy pracují s *pmInDom*, ovšem agent uchovává instance StatsD metrik pod klíčem, který je daný tagy daného datagramu ve formátu *JSON*. Existuje za účelem urychlení exekuce obslužných funkcí.

size_t pcp_instance_domain_count
: Počet prvků v **pmdaIndom* pcp_instance_domains**.

size_t pcp_metric_count
: Počet prvků v **pmdaMetric* pcp_metrics**.

size_t pcp_hardcoded_metric_count
: Počet statických metrik: interní statistiky agenta a metriky reprezentující aktuální konfiguraci.

size_t pcp_hardcoded_instance_domain_count
: Počet statických domén instancí: domény instancí pro statické metriky a výchozí domény pro dynamické metriky.

size_t generation
: Identifikátor naposledy synchronizované generace StatsD metrik.

int notify

Bitová mapa sloužící k předání notifikace PCP. Při synchronizaci nové generace StatsD metrik agent pomocí této hodnoty o tom faktu notifikuje PMCD.

```
1  // pmdastatsd.h
2  extern struct pmda_data_extension {
3     struct agent_config* config;
4     struct pmda_metrics_container* metrics_storage;
5     struct pmda_stats_container* stats_storage;
6     pmdaMetric* pcp_metrics;
7     pmdaIndom* pcp_instance_domains;
8     pmdaNameSpace* pcp_pmns;
9     dict* instance_map;
10    size_t pcp_instance_domain_count;
11    size_t pcp_metric_count;
12    size_t pcp_hardcoded_metric_count;
13    size_t pcp_hardcoded_instance_domain_count;
14    size_t generation;
15    int notify;
16 } pmda_data_extension;
```

Zdrojový kód 14: Pomocná data předaná každé obslužné funkci

Tato uživatelem definovaná data již ale nejsou předána funkcím, které slouží k předání hodnoty, případně popisku konkrétní metriky. V případě *statsd_fetch_callback* funkce lze využít první parametr, kterým je struktura *pmdaMetric*. Ta podobně jako *pmdaExt* obsahuje hodnotu, kterou může uživatel využít k držení libovolných dat. [4] Agent zde tedy uloží strukturu *pmda_metric_helper*, která mimo jiné také obsahuje referenci na již zmíněnou *pmda_data_extension*.

```
1  // pmdastatsd.h
2  extern struct pmda_metric_helper {
3     struct pmda_data_extension* data;
4     const char* key;
5     struct metric* item;
6  } pmda_metric_helper;
```

Zdrojový kód 15: Pomocná data přidaná ke každé instanci struktury pmdaMetric

Pro získání kontextu *pmda_data_extension* ve funkci *statsd_label_callback* je třeba se uchýlit ke globálním proměnným. [24] Volání funkce *statsd_label_callback* vždy předchází volání funkce *statsd_label*, kde si tedy můžeme uložit referenci pro pozdější použití.

[24] Globální v rámci souboru.

```
1 // pmda-callbacks.c
2 static pmdaExt* g_ext_reference;
3 int
4 statsd_label(int ident, int type, pmLabelSet** lp, pmdaExt* pmda) {
5    g_ext_reference = pmda;
6    statsd_possible_reload(pmda);
7    return pmdaLabel(ident, type, lp, pmda);
8 }
```

Zdrojový kód 16: Uložení kontextu v globální proměnné pro pozdější využití funkcí statsd_label_callback

S těmito pomocnými strukturami a ostatními předanými parametry mají zpětná volání *statsd_label_callback* a *statsd_metric_callback* vše, co potřebují, aby dokázaly najít hodnoty metrik, včetně jejich instancí, a hodnoty dále předat.

Synchronizace mezi interní a PCP reprezentací StatsD metrik je prováděna na základě komparace hodnot *pmda_data_extension->metrics_storage->generation* (generace interních metrik) a *pmda_data_extension->generation* (generace aktuální PCP reprezentace). Liší-li se generace, je PCP reprezentace metrik regenerována na základě interní reprezentace. Jako součást tohoto procesu se vytvoří nový jmenný prostor metrik, jimž se dodá sémantika na základě jejich typů, aktualizují se doménové instance, otagované hodnoty metrik se mapují na instance metrik a o této skutečnosti je PCP notifikováno. Jedná se o relativně rozsáhlý proces, při jehož dokončení je číslo generace PCP reprezentace aktualizováno, a nedojde-li k přidání nové metriky, případně otagované hodnoty metriky či k jejímu odebrání, což v současnosti nemůže uživatel vyvolat, nebude synchronizace znovu provedena.

5 Testy

Funkčnost agenta je důkladně pokryta testy, které byly integrovány do již existující velmi rozsáhlé testovací sady PCP, jež se automaticky spouští mimo jiné při každém pull-requestu [25] a zajišťuje stabilitu a spolehlivost kódu napříč verzemi PCP v mnoha prostředích, která PCP podporuje. [1] [15]

Během vývoje agenta byly průběžně prováděny unit testy a integrační testy síťové komunikace, skriptované v Bashi. Některé integrační testy byly průběžně přepsány do jazyka Ruby. Po integraci agenta do PCP repozitáře byly dosavadní integrační testy přepsány na doporučení udržovatelů PCP do kombinace Bashe a Pythonu. Toto poslední přepsání bylo z důvodu větší obeznámenosti komunity PCP s Pythonem, a tudíž pro ně snadnější údržby, a to obzvláště s přihlédnutím k

[25] Tento fakt lze pozorovat na libovolném pull-requestu v repozitáři PCP, například: https://github.com/performancecopilot/pcp/pull/1731.

faktu, že testovací sada PCP již disponuje vlastními testy napsanými v Pythonu.

5.1 Unit testy

V počátku práce byly během implementace parserů vytvořeny jednoduché unit testy za účelem zaručení korektního zpracování textu na vstupu do struktur, které interně reprezentují StatsD metriky. V pozdější fázi vývoje byly tyto testy nahrazeny testy integračními. Tyto testy a jejich orchestraci lze stále nalézt v původním repozitáři github.com/Erbenos/pmdastatsd. Cesty této kapitoly budou referovat k tomuto repozitáři.

Unit testy byly používány výhradně během vývoje před integrací do PCP repozitáře, který má k testům jiný přístup. Vzhledem k tomu, že práce na agentovi v původním repozitáři přestaly někdy před létem 2020, je kód zastaralý, a byl tudíž archivován. Při spouštění je třeba použít starší verzi překladače *gcc*, maximálně *gcc 9.0*.

Implementace unit testů je součástí souborů jednotlivých parserů. Soubory `src/parser-ragel.rl` pro *RAGEL* parser a `src/parser-basic.c` pro *BASIC* parser mají svou vlastní funkci *main*. Tyto funkce jsou však přítomny pouze tehdy, je-li soubor zkompilován se specifickými vlajkami v závislosti na tom, o jaký parser se jedná. Spuštění testů odpovídá spuštění agenta s těmito funkcemi *main*.

Součástí konfigurace souboru `GNUMakefile` jsou příkazy *test-ragel* a *test-basic*, které zkompilují dané parsery a spustí test.

Unit testy umožnily snadno iterovat nad vlastnostmi jednotlivých parserů bez obav, že by některý z nich nebyl schopen zpracovat řetězce či že by se výsledky zpracování lišily napříč parsery.

5.2 Integrační testy

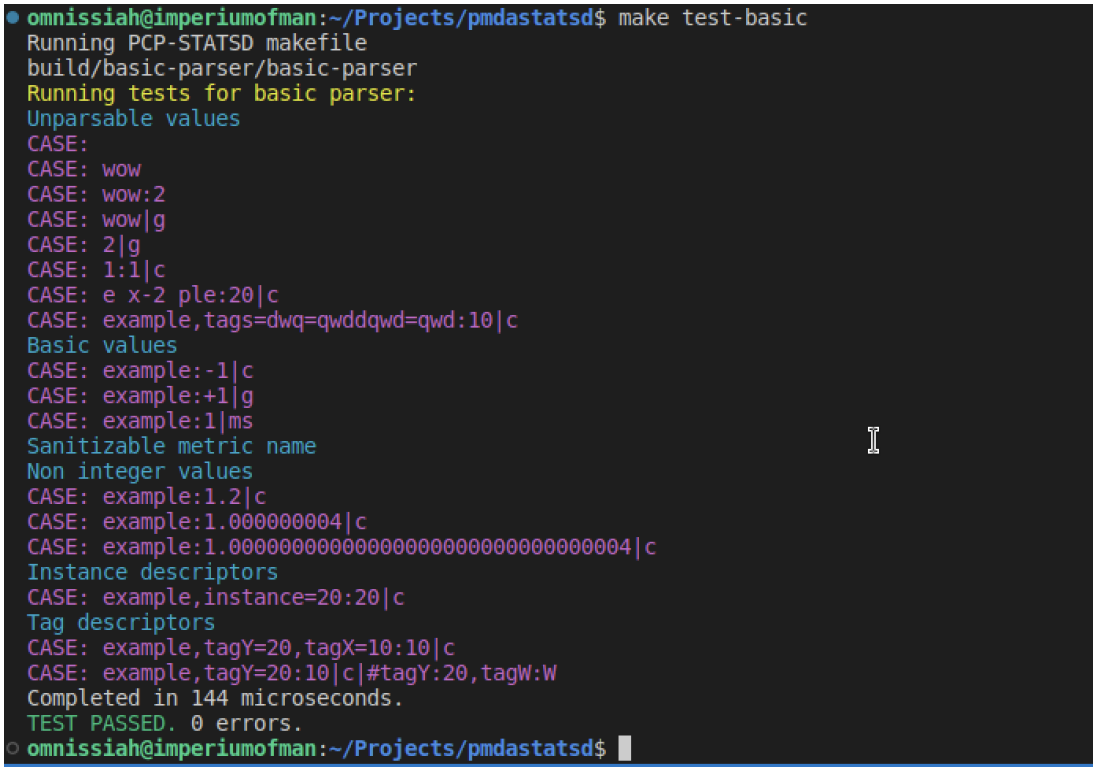
Integrační testy ověřují, zda agent funguje korektně při běžném použití napříč všemi úrovněmi abstrakce, ať už se jedná o korektní zpracování konfigurace agenta, chování jednotlivých typů metrik a jejich popisků, o integraci s PCP nebo o práci s pamětí a sítí.

5.2.1 Orchestrace testů v PCP

Soubory související s testy se nacházejí v podsložce `qa` PCP repozitáře, všechny cesty k souborům v rámci této kapitoly uvažujte relativně vůči této složce.

Podrobnosti o tom, jak se testy orchestrují, jsou popsány v souboru `README`. Vzhledem k množství a komplexitě testů v celé testovací sadě PCP se zde zaměříme pouze na základní informace nutné k práci s testy pro agenta. [16]

Soubory s testy jsou pojmenované `NNNN`, kde `N` je číslo testu (0 v prefixu mohou být vynechány). Tyto soubory jsou zpravidla Bash skripty. Každý soubor má do páru soubor `NNNN.out`, který slouží jako „golden-file"- očekávaný výstup



Obrázek 19: Ukázka zúženého výstupu spuštění unit testů pro BASIC parser

vygenerovaný spuštěním souboru `NNNN`. Pokud je výstup shodný, pak daný test prošel. V opačném případě test selhal. [16]

Vzhledem k velkému množství testů a konvenci jejich pojmenování jsou testy dále kategorizovány do skupin, kdy jeden test může být součástí mnoha skupin. Toto mapování je dáno souborem `group`. [16] Všechny testy agenta jsou součástí skupiny *pmda.statsd*. Tedy testy:

`1598` Instalace a odebrání agenta

`1599` Instalace a odebrání agenta, kontrola počtu statických metrik a počtu jejich hodnot

`1708` Zpracování konfiguračního souboru, dotázání se na konfiguraci

`1709` Výchozí hodnoty a existence statických metrik

`1710` Aktualizace statických metrik nesouvisejících s konfigurací

`1711` Chování metrik typu *counter*

`1712` Chování metrik typu *gauge*

`1713` Chování metrik typu *duration*

1714 Chování otagovaných hodnot metrik

1715 Chování agregace metrik typu *duration* pomocí HDR_histogramu

1716 Test stability jmenného prostoru metrik agenta

1717 Schopnost agenta naslouchat na konfigurovaném portu

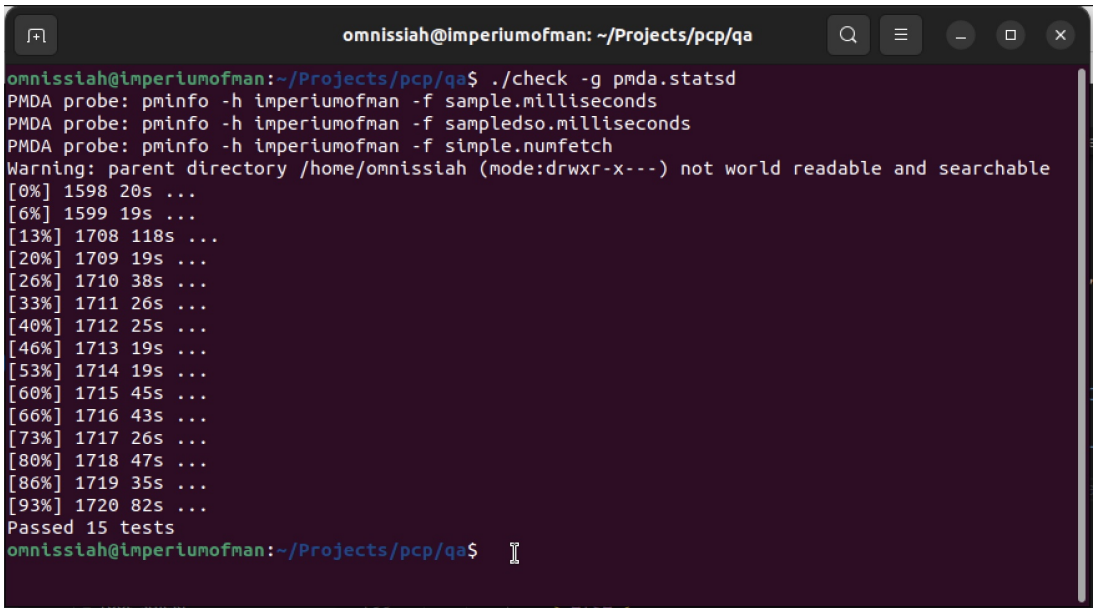
1718 Chování jednotlivých úrovní specifičnosti logování

1719 Konfigurovatelnost maximální velikosti datagramu

1720 Práce s pamětí, detekce úniků paměti pomocí nástroje Valgrind

Všechny výše uvedené testy používají funkce a proměnné, které jsou definovány v `common*` souborech jako například *_prepare_pmda*, *_check_valgrind* atd., jež usnadňují skriptování. S výjimkou testu 1598 testy zajistí prostředí pro exekuci Python skriptů, které interagují s PCP, sítí a agentem za účelem ověření domény, již má daný test pokrýt. Samy o sobě jsou bash skripty relativně hloupé. Python testy se nacházejí ve složce `statsd/src`. Bash soubory s testy tedy spustí Python skripty, které jsou vzájemně mapovány 1 : 1. Například test `1599` spustí `statsd/src/qa/cases/01.py`. Všechny Python testy sdílí skupinu funkcí pro interakci s nástroji PCP, sítí a agentem, které se nachází v `statsd/src/utils/pmdastatsd_test_utils.py`.

Jednotlivé testy lze spustit příkazem `check NNNN`. Skupinu testů (dle příslušenství ke skupině definované v souboru `groups`) lze spustit pomocí `check -g xxx`. Všechny testy jsou spuštěny pouhým `check`.



Obrázek 20: Spuštění integračních testů pmdastatsd

5.2.2 Ostatní

Jak již bylo zmíněno, PCP sada testů se automaticky spouští při různých událostech, jako je například otevření pull-requestu. Vývoj zpočátku probíhal v separátním repozitáři, kde byla také automatizace, ale na nižší úrovni. To bylo dostačující, aby se zaručilo, že agent je sestavitelný, spustitelný a testy jsou splněny. Pro tuto automatizaci byl použitý nástroj Travis-CI. Konfigurace této automatizace je dána souborem `.travis.yml` v původním repozitáři agenta.

5.2.3 Příprava prostředí pro exekuci testů

Vzhledem ke komplexitě a rozsahu testovací sady PCP je třeba před prací s testy instalovat sadu závislostí. Distribuce PCP, které slouží pouze k běhu PCP, neobsahují tyto závislosti. Postup toho, jak závislosti nainstalovat a nastavit prostředí tak, aby bylo možné testy spustit, je popsán v kapitolách "6.3 Instalace ze zdrojů" a "6.4 Sestavení a instalace PCP s pmdastatsd". Případně lze využít již zprovozněné prostředí reprezentované virtuálním strojem, který je součástí přiloženého média.

6 Ladění

Během implementace agenta bylo využito několika metod ladění programu.

6.1 GNU Project Debugger

GNU Project Debugger (díle jen 'gdb') umožní uživateli vidět *dovnitř* programu během jeho exekuce, případně odhalit, co program zrovna vykonával, když spadl. [17] Mezi hlavní kompetence gdb patří: [17]

- Spustit program v kontrolovaném prostředí.
- Vynutit pozastavení programu na základě předem definovaných podmínek.
- Zkoumat, co se stalo, když se program zastavil.
- Změnit hodnoty v programu za jeho chodu.

V rámci vývoje agenta byl gdb použit především v rámci integrace s C rozšířením pro textový editor Visual Studio Code. [26] Agent byl během vývoje zkompilován pomocí *gcc* s vlaječkou *-g*, která zaručí, že se v rámci kompilace do programu přidají metadata, s jejichž pomocí se může gdb v programu zorientovat. Takto zkompilovaný agent umožňuje propojení s gdb za jeho běhu. Potom lze program libovolně krokovat v jakémkoliv z jeho vláken, což je při ladění k nezaplacení.

[26], a to hlavně kvůli grafickému rozhraní pro krokování, inspekci a editaci hodnot proměnných.

6.2 dbpmda

PCP má svůj specifický nástroj pro testování interakcí agenta s PMCD. dbpmda je utilita pro zkoušení a testování chování agenta, mimo jiné simulující volání obslužných funkcí dotazů PMCD, umožňující ladit tuto integraci. [18]

6.3 Valgrind

Valgrind je nástroj pro ladění a profilování linuxových programů. S Valgrindem je možné automaticky detekovat chyby související se správou paměti. [19]

Společně s dbpmda byl Valgrind použit k verifikaci korektnosti práce s pamětí. Integrační testy využívají Valgrind, kdy Python skript spustí dbpmda pod Valgrindem a agent se nechá pracovat na datagramech reprezentujících jeho obvyklé využití. Poté, co jsou všechny datagramy zpracovány, je simulováno čtení těchto hodnot ze strany PCP. Nakonec se agentovi zašle signál `SIGINT`, který agent interpretuje jako pokyn ke svému ukončení. Agent po sobě "uklidí", uvolní všechnu paměť, která byla během chodu programu alokována, a ukončí se. V tomto okamžiku je ukončen i Valgrind, který vypíše informace týkající se paměťové stopy agenta, kde můžeme např. vyčíst to, že po ukončení agenta nezůstala dynamicky alokovaná paměť neuvolněna. [27]

6.4 NetCat

NetCat (dále jen 'nc') je utilita pro práci se sítí. Snadno umožňuje uživateli např. zaslat libovolné datagramy na libovolný port [20], což bylo velmi často využito během vývoje agenta. Například pro zaslání datagramu na adresu 127.0.0.1:8125 může uživatel použít nc následovně:

```
echo "user_login:1337|ms" | nc -u -w0 127.0.0.1 8125
```

Dřívější podoby integračních testů využívaly nc pro simulaci provozu sítě podobným způsobem. Později byl nc použit pouze v rámci testování během vývoje, po přepsání testů do Pythonu byly použity moduly Pythonu, které umožňují prakticky to samé jako nc, ovšem ergonomičtěji. [21]

7 Návod pro instalaci PCP

V návodu se uvažuje čistá instalace distribuce Ubuntu 22.04.2 LTS (Jammy Jellyfish) [28].

[27] Popsaný test se nachází v souboru `qa/1720.out` repozitáře PCP.

[28] Distribuce je volně dostupná na adrese https://releases.ubuntu.com/jammy/ubuntu-22.04.2-desktop-amd64.iso.

7.1 Přiložené DVD

Přiložené médium obsahuje obraz virtuálního stroje výše uvedené distribuce ve formátu `.ova` a jenž obsahuje již nainstalované PCP včetně StatsD agenta ze zdrojových kódů společně se všemi závislostmi nutnými pro spuštění integračních testů agenta. Pro přihlášení na systému lze použít uživatele "omnissiah" a heslo "1337".

7.2 Instalace z repozitářů distribucí

Organizace PCP udržuje balíčky pro nemálo distribucí a operačních systémů, jejichž dostupnost je popsaná na adrese https://pcp.io/download.html.

7.3 Instalace ze zdrojů

Nejprve je třeba si stáhnout zdrojové kódy PCP. Agent je jejich součástí, není třeba jej stahovat zvlášť.

Zdroje PCP lze stáhnout z git repozitáře, ovšem minimální instalace Ubuntu nedisponuje gitem, tudíž je třeba si jej stáhnout separátně.

```
sudo apt install git
```

Následně vytvoříme adresář ve kterém provedeme všechny další úkony:

```
cd ~
mkdir Projects
```

Naklonujeme PCP:

```
git clone https://github.com/performancecopilot/pcp.git
```

Dále je nezbytné si stáhnout několik balíčků, které jsou třeba pro sestavení, konfiguraci sestavení, či se jedná o závislosti agenta potřebné pro jeho vývoj.

7.3.1 Balíčky nutné pro sestavení

Mnohé distribuce obsahují některé tyto balíčky již v základu, testovací systém ovšem nikoliv. Příkaz pro nainstalování:

```
sudo apt install zlib1g-dev autoconf libtool make cmake
```

7.3.2 Ragel

Už bylo zmíněno, že parser je distribuován již zkompilovaný, Ragel se ovšem přesto může hodit za účelem vývoje či generace vizualizací. Příkaz pro nainsta-

lování:

```
sudo apt-get install ragel
```

7.3.3 chan

Instalace spočívá v naklonování repozitáře, konfiguraci sestavení, samotné instalaci a nutné regeneraci cache dynamického linkeru:

```
git clone https://github.com/tylertreat/chan.git
cd chan
./autogen.sh && autoreconf -f -i && ./configure
sudo make install
sudo ldconfig
```

7.3.4 HdrHistogram_c

Provedení prakticky stejných kroků jako v případě Ragelu:

```
git clone https://github.com/HdrHistogram/HdrHistogram_c
   .git
cd HdrHistogram_c
cmake .
sudo make install
sudo ldconfig
```

7.3.5 Závislosti PCP

Všechny následující sekvence příkazů budou provedeny relativně vůči adresáři PCP. Přesuneme do něj:

```
cd ~/Projects/pcp
```

Před sestavením PCP je dobré se ujistit, že na systému je dostupný potřebný software. PCP disponuje skriptem `qa/admin/check-vm`, který si projde systém hosta a na základě heuristik dokáže napovědět jména balíčků závislostí, pod nimiž je pravděpodobně dostupný potřebný software pro danou distribuci hosta. Ten společně se správcem balíčků použijeme pro instalaci závislostí. [22]

Níže uvedený příkaz postupně instaluje závislosti jednu po druhé odděleně[29],

[29]Na dané verzi Ubuntu je po instalaci balíčku zobrazen dialog, který požaduje vstup od uživatele blokující tento proces. Je v zájmu uživatele tento dialog automa-

ovšem uživatel by měl nejprve zkontrolovat seznam balíčků na výstupu *qa/admin/check-vm -fp* a zvážit, zda neinstaluje nežádoucí software.

```
qa/admin/check-vm -fp | xargs -n 1 sudo apt-get install
   -y
```

Jakmile skript instalující balíčky skončí, znovu spustíme kontrolu chybějících balíčků:

```
qa/admin/check-vm -fp
```

Žádoucí by bylo, aby byl vstup skriptu prázdný, ovšem na našem systému dostaneme následující:

```
kubernetes-client libqt4-dev libsoqt-dev libsoqt4-dev
```

Ve výchozích repozitářích Ubuntu balíčky s danými jmény nejsou a PCP nemůže smysluplně držet seznam všech možných balíčků na všech možných platformách, které podporuje, pro všechny možné distribuce a verze, tudíž se spoléhá pouze na heuristiky. Je na uživateli, aby si obstaral tyto závislosti pro svou platformu. Pro výše uvedenou distribuci Ubuntu lze postupovat následovně. Balíček *kubernetes-client* nahradíme Snapem *kubectl*:

```
snap install kubectl --classic
```

S balíčky souvisejícími s Qt4 to bude trochu složitější, od verze 20.04 byly Qt4 balíčky odebrány z oficiálních repozitářů [30] Proto repozitáře obsahující tyto balíčky pro Ubuntu 22.04 neexistují. Zatímco je možné, že by se podařilo sestavit software poskytovaný těmito balíčky manuálně, vzhledem k tomu, že nebudeme potřebovat žádné grafické rozhranní nástrojů, jež PCP poskytuje, můžeme absenci těchto balíčků akceptovat.

Nyní jsou nainstalované všechny nezbytné závislosti pro chod PCP. Pokračujeme založením uživatele, pod kterým klíčové části instalace PCP poběží: [22]

```
sudo -s
groupadd -r pcp
useradd -c "Performance Co-Pilot" -g pcp -d /var/lib/pcp
    -M -r -s /usr/sbin/nologin pcp
exit
```

ticky přeskočit pomocí specifikace environmentální proměnné `export DEBIAN_FRONTEND =noninteractive` před spuštěním samotného příkazu.

[30] Ubuntu je distribuce založená na distribuci Debian [23] a ta již měla toto v plánu od roku 2015 [24].

Posléze nakonfigurujeme sestavení a nainstalujeme: [22]

```
./configure --prefix=/usr --libexecdir=/usr/lib --
   sysconfdir=/etc --localstatedir=/var --with-rcdir=/etc
   /init.d
make
sudo -s
make install
exit
```

Pokud proběhlo všechno bez problému, stačí spustit PMCD: [22]

```
sudo systemctl start pmcd
```

Nyní je PCP úspěšně nainstalován a monitorující nástroje se mohou dotazovat na metriky. Toto lze snadno zjistit např. vypsáním všech dostupných metrik:

```
pminfo
```

Lze vidět, že ještě nejsou dostupné žádné StatsD metriky, protože příkaz nevypsal žádné s prefixem "statsd.". Agent sice byl nainstalován ve smyslu jeho kompilace a umístění do správného adresáře, ovšem není aktivní. Je nejprve třeba spustit jeho instalační skript.

Agenti jsou po instalaci PCP umístěni do adresáře daného proměnnou prostředí `$PCP_PMDAS_DIR`. Tato a další podobné proměnné jsou definovány v souboru `/etc/pcp.conf`. [22] Agenta tedy nainstalujeme:

```
. /etc/pcp.conf
cd $PCP_PMDAS_DIR/statsd
sudo ./Install
```

Posledním krokem je vypsání všech StatsD metrik:

```
pminfo statsd
```

Vrátil-li příkaz neprázdný seznam metrik, agent byl úspěšně nainstalován.

8 Uživatelská příručka

Příručka uvažuje instalaci PCP a agenta dle kroků popsaných v předchozí kapitole. Velmi podobná verze je dostupná v angličtině v souborech `README.md` a `pmdastatsd.1` adresáře agenta.

8.1 Funkce

- Metriky typu *counter*
- Metriky typu *gauge*
- Metriky typu *duration* s instancemi:
  - Minimum
  - Maximum
  - Medián
  - Průměr
  - 90. percentil
  - 95. percentil
  - 99. percentil
  - Množství
  - Směrodatná odchylka
- Parsování zpráv buďto *RAGEL* parserem, nebo *BASIC* parserem
- Agregace *duration* metrik HDR Histogramem, případně naivním algoritmen
- Popisky
- Logování
- Statistiky o agentu samotném
- Konfigurace
  - `.ini` soubor
  - Parametry příkazové řádky

8.2 Instalace

V adresáři agenta je přítomen `Install` skript, který agenta nainstaluje a spustí.

8.3 Odstranění

V adresáři agenta se také nachází `Remove` skript, který agenta deaktivuje. Soubory zůstanou přítomny. Chcete-li agenta odebrat úplně, stačí smazat adresář `$PCP_PMDAS_DIR/statsd`.

8.4 Konfigurace agenta

Agenta je možno konfigurovat jak pomocí konfiguračního souboru, tak parametry příkazové řádky. Vzhledem k tomu, že správu agentů (včetně jejich spouštění) orchestruje PMCD, ve valné většině případů bude ideální použít konfigurační soubor, ale podpora pro argumenty příkazové řádky je přítomna, pokud je to třeba.

8.4.1 Ini soubor

Agent při spuštění hledá ve svém adresáři konfigurační soubor *pmdastatsd.ini*. Podporuje následující konfigurační parametry:

max_udp_packet_size
: Maximální povolená velikost paketu.
Výchozí hodnota: **1472**

port
: Port, na kterém má agent naslouchat, pro příchozí data.
Výchozí hodnota: **8125**

verbose
: Úroveň upovídanosti výstupu agenta do logů. Podporované hodnoty jsou čísla od 0 do 2 včetně. Každá vyšší úroveň zahrnuje i tu nižší.
0 = Vypíše použitou konfiguraci, stav socketu ke čtení a prvních 100 zahozených zpráv.
1 = Zobrazí informace o synchronizaci StatsD metrik do jejich PCP reprezentace.
2 = Nejvyšší úroveň, vypíše informaci o všech zahozených zprávách.
Výchozí hodnota: **0**

debug_output_filename
: Jméno souboru, do kterého agent při přijetí SIGUSR1 signálu vypíše informace o všech agregovaných metrikách.
Výchozí hodnota: **debug**.

version
: Vlaječka určující, zda má agent při startu vypsat do logů svou verzi.
Výchozí hodnota: **0**

parser_type
: Algoritmus má být použit pro parsování příchozích zpráv. Podporované hodnoty jsou:
0 = *BASIC* parser
1 = *RAGEL* parser
Výchozí hodnota: **0**

duration_aggregation_type
Agregační algoritmus pro metriky typu *duration*.
0 = naivní
1 = HDR Histogram.
Výchozí hodnota: **1**

max_unprocessed_packets
Maximální počet datagramů, které agent může držet v jednotlivých kanálech sloužících k mezivláknové komunikaci.
Výchozí hodnota: **2048**

8.4.2 Parametry příkazové řádky

Agent akceptuje argumenty příkazové řádky, které musí podporovat všichni ostatní agenti, včetně parametrů podporovaných konfiguračním souborem:

- **--max-udp, -Z**
- **--port, -P**
- **--verbose, -v**
- **--debug, -g**
- **--debug-output-file, -Z**
- **--version, -s**
- **--parser-type, -r**
- **--duration-aggregation-type, -a**
- **--max-unprocessed-packets-size, -z**

V případě, kdy je argument předán jak pomocí konfiguračního souboru, tak parametrem příkazové řádky, hodnota předaná příkazovou řádkou má prioritu.

8.5 Použití

Agent naslouchá na specifikovaném portu pro jakýkoliv obsah ve formě:

```
<nazev-metriky>:<hodnota>|<typ>
```

V jednom datagramu může být přitomno více takovýchto zpráv rozdělených znakem nového řádku, a to následovně:

```
<nazev-metriky>:<hodnota>|<typ>\n<nazev-metriky>:<
   hodnota>|<typ>
```

Kdy jednotlivé sekce musí odpovídat následujícím sledům znaků:

- `nazev-metriky` = $[a-z][a-zA-Z0-9\_.]*$
- `hodnota` = popsáno v sekcích níže
- `typ` = $c|g|ms$

Všechny zaznamenané metriky budou dostupné pod jmenným prostorem "statsd.*". Jakmile jednou agent přijme pro daný `nazev-metriky` konkrétní typ, bude zahazovat všechny zprávy pro danou metriku, které mu neodpovídají.

8.5.1 Metriky typu counter

Hodnoty typu čítač. Všechny příchozí hodnoty jsou postupně akumulovány.

```
<nazev-metriky>:<hodnota>|c
```

Kde *hodnota* je kladné číslo. Po agregaci následujících zpráv:

```
metric:20|c
metric:10|c
metric:3.3|c
```

Bude výsledná hodnota:

```
pminfo -f statsd.metric
-> inst[0 or "/"] value 33.3
```

8.5.2 Metriky typu gauge

Libovolně měnitelné hodnoty. Zprávy mohou specifikovat jak přímé nastavení, tak inkrement či dekrement stávající hodnoty.

```
<nazev-metriky>:<hodnota>|g
```

Kdy `hodnota` může nabývat:

- `-inkrement` - agent odečte předanou hodnotu od hodnoty stávající
- `+dekrement` - agent sečte předanou hodnotu s hodnotou stávající
- `nastavení` - agent nastaví metriky na předanou hodnotu

Výchozí hodnota pro metriky typu *gauge* je **0**. Po agregaci následujících zpráv:

```
metric:20|g
metric:+10|g
metric:-3.3|g
```

Bude výsledná hodnota:

```
pminfo -f statsd.metric
-> inst [0 or "/"] value 26.7
```

8.5.3 Metrika typu duration

Agreguje hodnoty způsobem umožňujícím inspekci četnosti hodnot a dalších metadat dané metriky.

```
<nazev-metriky>:<hodnota>|ms
```

Kdy `hodnota` je kladné číslo. Po agregaci následujících zpráv:

```
metric:10|ms
metric:20|ms
```

Bude výsledná hodnota:

```
pminfo -f statsd.metric
->
inst [0 or "/min"] value 10
inst [1 or "/max"] value 20
inst [2 or "/median"] value 10
inst [3 or "/average"] value 15
inst [4 or "/percentile90"] value 20
inst [5 or "/percentile95"] value 20
inst [6 or "/percentile99"] value 20
inst [7 or "/count"] value 2
inst [8 or "/std_deviation"] value 5
```

8.5.4 Popisky

Datagramy také mohou obsahovat tagy ve formátu `klíč:hodnota`, oddělené čárkou, následovně:

```
metric,tagX=X,tagW=W:5|c
```

Nebo:

```
metric:5|c|#tagX:X,tagW:W
```

Kde platí:

- `tagX` je *klíč*, `X` je *hodnota*
- `tagW` je *klíč*, `W` je *hodnota*

Jak klíč, tak hodnota tagu musí splňovat `[a-zA-Z0-9_.]{1,}`. Oba formáty mají stejnou výpovědní hodnotu a mohou být kombinovány. Pokud se *klíč* nachází ve více než jednom tagu, přednost bude mít *hodnota* nejblíže konci řetězce. Následující zpráva je validní:

```
metric,tagX=1:5|c|#tagX:2
```

Tag s *klíčem* `tagX` bude mít *hodnotu* `2`. Tyto tagy jsou použity pro mapování hodnot na PCP instance. Tagy se promítnou do PCP popisku instance, kde jsou seřazeny dle *klíče*.

Datagram s jedním popiskem:

```
metric,tagX=X:5|c
```

Tato zpráva se mapuje do PCP následovně:

```
pminfo -f --labels statsd.metric
->
inst [0 or "/tagX=X"] value 5
inst [0 or "/tagX=X"] labels {"tagX":"X"}
```

Lze pracovat se zprávami, které obsahují více popisků. Pokud je popisků více, jsou v názvu instance odděleny pomocí řetězce "::". Příklad:

```
metric,tagX=X,tagW=W:5|c
```

Toto bude vyhodnoceno na:

```
pminfo -f --labels statsd.metric
->
inst [0 or "/tagX=X::tagW=W"] value 5
inst [0 or "/tagX=X::tagW=W"] labels {"tagX":"X","tagW
   ":"W"}
```

Metriky typu *duration* jsou mapovány na více instancích i bez zaslání jakýchkoliv tagů. Zpracováním zprávy s tagy pro již existující metriku *duration* bez popisků

vytvoří dalších 9 instancí, právě tolik má totiž každá metrika typu *duration* pevně definovaných instancí. Příklad:

```
metric:200|ms
metric:100|ms
metric:200|ms
metric,target=cpu0:10|ms
metric,target=cpu0:100|ms
metric,target=cpu0:1000|ms
```

Zpracování těchto zpráv vytvoří 18 instancí metriky `metric`, s názvy jsou složeny následovně:

```
pminfo -f --labels statsd.metric
->
...
inst [10 or "/max::target=cpu0"] value 1000
inst [10 or "/max::target=cpu0"] labels {"target":"cpu0
   "}
...
```

8.5.5 Pevně definované metriky

Agent exportuje několik metrik o sobě samém, které jsou hned po startu přístupné PCP:

- **statsd.pmda.received** - Počet přijatých zpráv
- **statsd.pmda.parsed** - Počet parsovaných zpráv
- **statsd.pmda.dropped** - Počet zahozených zpráv
- **statsd.pmda.aggregated** - Počet agregovaných zpráv
- **statsd.pmda.metrics_tracked** - Počet sledovaných metrik podle typu
  - **counter** - Metriky typu *counter*
  - **gauge** - Metriky typu *gauge*
  - **duration** - Metriky typu *duration*
  - **total** - Počet sledovaných metrik celkem
- **statsd.pmda.settings.max_udp_package_size** - Maximální velikost packetu
- **statsd.pmda.settings.verbose** - Nastavení **verbose** vlaječky

- **statsd.pmda.settings.debug_output_filename** - Hodnota konfiguračního nastavení **debug_output_filename**
- **statsd.pmda.settings.port** - Port, na který agent naslouchá
- **statsd.pmda.settings.parser_type** - Hodnota konfiguračního nastavení **parser_type**
- **statsd.pmda.settings.duration_aggregation_type** - Hodnota konfiguračního nastavení **duration_aggregation_type**

Tato jména nejsou dostupná pro použití uživatelem. Zprávy s těmito názvy metrik nebudou zpracovány. Celý jmenný prostor *statsd.pmda.** není doporučen pro použití uživatelem.

9 Možná zlepšení

9.1 Granulární konfigurace velikosti bufferů

Vlákna mezi sebou komunikují pomocí kanálů, ty ovšem mají nyní pevně danou velikost bufferu, odpovídající parametru konfigurace **max_unprocessed_packets**. Velikost tohoto bufferu je pro všechny kanály stejná. Stojí za zvážení, zda by se nevyplatilo mít tuto konfiguraci více granulární vzhledem k rozdílné komplexitě práce, kterou vlákna dělají.

9.2 Odebrání BASIC parseru

V porovnání s *RAGEL* parserem *BASIC* parser je mnohem těžší na údržbu a rozšíření. Parser není možné snadno vizualizovat. Ohlédnu-li se zpátky, mít dva parsery nepřidává agentovi žádnou hodnotu, pouze to věci zbytečně komplikuje.

9.3 Podpora pro správu metrik běžícího agenta

Nyní nelze nijak interagovat s běžícím agentem, opomine-li integraci agenta s PCP samotným, umožňujícím dotazování se na hodnoty jednotlivých metrik a vypsáním všech zaznamenaných metrik, obdržením `SIGUSR1` signálu. Originální StatsD démon ovšem poskytuje jednoduché rozhraní pro administraci metrik, umožňující například smazání libovolné metriky. [25]

PCP má mechanismus pro zapsání hodnoty metriky uživatelem, reprezentován utilitou *pmstore* [31] [26], ale podpora pro toto chování není agentem implementována.

[31] Vzhledem k tomu, že hodnoty metrik jsou drženy samotnými agenty, *pmstore* pouze vygeneruje pokyn ke změně metriky, jak tento pokyn agent zpracuje, záleží na něm. [26] [27]

Podpora pro *pmstore* spočívá v pouhé implementaci obsluhy pro dotaz *store*. [28] Není ovšem zcela jasné, jak by se toto chovalo s ohledem na *duration* metriku, která má vždy více než jednu instanci. Při pokusu o zapsání hodnoty do *duration* by buď byly všechny instance vynulovány, nebo by se neprovedlo nic a tento fakt by byl zapsán do logů. Toto rozhraní by rovněž mohlo reflektovat chování, které se v současnosti provede v případě příchozího datagramu. V tomto případě by se vždy jednalo o změnu hodnoty stejným způsobem jako při zpracování příchozího datagramu.

9.4 Podpora pro změnu konfigurace za běhu agenta

Není obtížné si představit uživatele, který chce provést změny konfigurace za chodu agenta, ať už kvůli ladění, či jako reakci na neočekávané množství příchozích datagramů, kdy může být žádoucí zaznamenat větší objem zpráv. V současnosti je nutné agenta restartovat a přijít tak o všechny dosud naměřené metriky. PCP sice umožňuje vytvářet archivy z již naměřených hodnot [29], ale vzhledem k sémantice metrik StatsD, kdy se jedná především o metriky s interním stavem, který je při restartu efektivně resetován, toto moc nepomůže.

Uvědomíme-li si, že agent již nyní exportuje své aktuální nastavení v podobě PCP metrik, například *statsd.pmda.settings.port*, nabízí se relativně elegantní API v podobě již zmíněného zpětného volání *store*, jehož implementace by efektivně umožnila uživateli provést změny v nastavení pomocí *pmstore* utility. Například: `pmstore statsd.pmda.settings.port 8000` by provedlo změnu v portu, na kterém agent naslouchá.

9.5 Více statických metrik

Agent by mohl o sobě exportovat také informaci o tom, kolik zpráv je drženo v bufferu jednotlivých kanálů. Toto by mohlo sloužit společně s logy a podporou pro změnu konfigurace za běhu agenta ke snazšímu ladění konfigurace.

9.6 Podpora metriky typu Set

Původní StatsD démon podporuje metriky typu *set*, které reprezentují čítače unikátních hodnot. [30]

Jednoduchá implementace by spočívala v držení všech dosud obdržených zpráv daného typu v hashmapě, kde hodnota v datagramu by sloužila jako klíč. Na žádost o hodnotu dané metriky by šlo pouze o vrácení počtu hodnot v této hashmapě.

9.7 Podpora rozšíření StatsD protokolu

StatsD protokolu má různá neoficiální rozšíření. Jedním z nich je třeba DogStatsD, které podporuje více typů metrik v porovnání s původním StatsD démonem, tagy bez hodnot a mnohé další. [31]

Byť ne všechny funkcionality, ať už v původním StatsD, či v rozšíření DogStatsD, dávají úplně smysl v kontextu PCP, některé lze integrovat a dále je využít. Parser agenta by přinejmenším mohl přijmout datagramy ve formátu DogStatsD a buďto zpracovat ty jejich části, které jsou v kontextu PCP smysluplné, či je zahodit, ovšem s dodatečným logováním pro snadnější ladění.

9.8 Refaktorování

Kód agenta se mi zdá přijatelný, věřím ale, že je zde mnoho míst pro zlepšení, ať už co se týče komentářů, pojmenování některých proměnných či samotné architektury. Rád bych docílil větší separace mezi kódem, který je specifický pro metriky daného typu v různých částech aplikace: parsování, agregace a předávání hodnot metrik; podobně jako je zajištěno pro komunikaci s PCP, přes funkce reprezentující zpětná volání pro jednotlivé dotazy PMCD. [28] Nyní je tento kód separován hlavně prostřednictvím souborů, což je dostačující, ovšem s rostoucí komplexitou agenta, je zde prostor pro refaktorování. Vzhledem k dosavadní komplexitě agenta je současný stav dostačující, ovšem s rostoucím počtem funkcí je na zvážení provedení výše uvedených změn.

Závěr

Výsledkem práce je PCP agent pro protokol StatsD, včetně dokumentace a testů. Agent je vhodný pro sběr a záznam dat poslaných přes síť a pro logování údajů z uživatelských aplikací s využitím jedné z mnoha klientských knihoven StatsD. Ve spojení s nástroji pro vizualizaci dat, jako je Grafana, či s nástroji obsaženými v PCP lze efektivně monitorovat libovolné aplikace.

V práci jsem rozebral vícevláknovou architekturu programu, metodu předávání dat mezi vlákny, přístup k parsování dat, agregaci dat různých typů a integraci s PCP.

Program je důkladně testován skripty v Pythonu a Bashi. Korektnost práce s pamětí je validována nástrojem Valgrind. Dokumentace programu je dostupná ve formátu Markdown a prostřednictvím manuálových stránek. Práce může sloužit jako příklad vývoje low-level aplikací v jazyce C, případně jako příklad vývoje agentů pro PCP.

Jsem rád, že jsem mohl svou prací přispět open-source organizaci a současně se dostat do styku s profesionály z oboru a mít tak možnost čerpat z jejich znalostí a zkušeností. Během vývoje agenta mi byl dostupný mentoring Lukáše Zapletala a Nathana Scotta, který mi velmi pomohl, i přestože jsme museli překonávat nepraktické rozdílné časové zóny.

Jsem velmi vděčný organizaci PCP za možnost účasti na Google Summer of Code 2019 s implementací tohoto agenta. Byla to příjemná zkušenost, která vedla k pozdější mé účasti na jiném projektu, pro mě již ve více familiární doméně, ale se stejnou organizací v rámci Google Summer of Code 2020.

Conclusions

Result of the thesis is a PCP agent for protocol StatsD, including documentation and tests. The agent is suitable for collecting and recording of data sent over the network, using one of many StatsD client libraries. In combination with interactive visualization tools such as Grafana or with built-in PCP tools, agent presents an easy way to monitor any kind of application.

I analyzed the multi-threaded architecture of the program, the method of sharing data between threads, approach to data parsing, aggregation of data of various types of metrics and PCP integration.

Program is thoroughly tested using combination of Python and Bash scripts and correct memory management is validated with Valgrind. The documentation is provided in Markdown and man page formats. Thesis can serve as an example of low-level application development in language C, possibly also as an example of PCP agent development.

I am happy that I was able to contribute to the PCP open-source organization, get in touch with professionals in the field and have the opportunity to draw from their knowledge and experience. Mentorship of Lukas Zapletal and Nathan Scott was available to me during the agent's development. This was very helpful, even though difference in timezones was challenging at times.

I am very grateful to the PCP organization for allowing me to participate in Google Summer of Code 2019 with agent's implementation. It was a pleasant experience, that led to later participation in Google Summer of Code 2020, also with PCP organization, though this time in a more familiar domain.

10 Obsah přiloženého média

doc/

src/
Zdrojové kódy textu práce v LaTeX-u.

ragel parser fsm.svg
Vizualizace Ragel parseru.

dist/
Text práce ve formátu PDF.

src/

pcp/
Zdrojové kódy PCP.
Naklonovaný repozitář https://github.com/performancecopilot/pcp/.

src/pmdas/statsd/
Zdrojové kódy agenta.

qa/
Zdrojové kódy testovací sady PCP, včetně testů pro agenta.

pmdastatsd/
Zdrojové kódy, nyní již archivované verze agenta.
Naklonovaný repozitář https://github.com/Erbenos/pmdastatsd/.

vm/
Virtuální stroj ve formátu `.ova` s nainstalovaným PCP a StatsD agentem, společně se všemi závislostmi nezbytnými pro jeho prezentaci, včetně spuštění testů.

Literatura

[1] *Performance Co-Pilot*. [online]. [cit. 2022-12-20].
Dostupný z: ⟨https://github.com/performancecopilot/pcp/⟩.

[2] *PCP Presentation*. [online]. [cit. 2023-3-13].
Dostupný z: ⟨https://pcp.io/papers/pcp-presentation/⟩.

[3] *'3. PMAPI–The Performance Metrics API — pcp 6.0.4-1 documentation'*. [online]. [cit. 2023-3-14].
Dostupný z: ⟨https://pcp.readthedocs.io/en/latest/PG/PMAPI.html⟩.

[4] *PCP Documenation - 2. Writing A PMDA*. [online]. [cit. 2022-12-24].
Dostupný z: ⟨https://pcp.readthedocs.io/en/latest/PG/WritingPMDA.html⟩.

[5] *pmdagetoptions(3) - Linux manual page*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨https://man7.org/linux/man-pages/man3/pmdagetoptions.3.html⟩.

[6] *PCP Documentation - 2.2. Performance Metrics Collection Daemon*. [online]. [cit. 2022-12-24].
Dostupný z: ⟨https://pcp.readthedocs.io/en/latest/UAG/InstallingAndConfiguringPcp.html#performance-metrics-collection-daemon-pmcd⟩.

[7] *PCP Documentation - 1.1. PCP Architecture*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨https://pcp.readthedocs.io/en/latest/PG/ProgrammingPcp.html#pcp-architecture⟩.

[8] *StatsD*. [online]. [cit. 2022-12-20].
Dostupný z: ⟨https://github.com/statsd/statsd⟩.

[9] *StatsD Backend*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨https://github.com/statsd/statsd/blob/master/docs/backend.md⟩.

[10] *HDR Histogram: A High Dynamic Range Histogram*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨http://hdrhistogram.org/⟩.

[11] *pcp-mmvstatsd/main.go at master · lzap/pcp-mmvstatsd · GitHub*. [online]. [cit. 2023-4-17].
Dostupný z: ⟨https://github.com/performancecopilot/pcp/tree/main/src/pmdas/json⟩.

[12] *Pure C implementation of Go channels*. [online]. [cit. 2021-7-26].
Dostupný z: ⟨https://github.com/tylertreat/chan⟩.

[13] *Ragel State Machine Compiler*. [online]. [cit. 2021-7-26].
Dostupný z: ⟨http://www.colm.net/open-source/ragel/⟩.

[14] *Graphviz Commands*. [online]. [cit. 2022-12-29].
Dostupný z: ⟨https://graphviz.org/doc/info/command.html⟩.

[15] *Performance Co-Pilot - Download*. [online]. [cit. 2022-12-30].
Dostupný z: ⟨https://pcp.io/download.html⟩.

[16] *Performance Co-Pilot - QA*. [online]. [cit. 2022-12-31].
Dostupný z: ⟨https://github.com/performancecopilot/pcp/blob/main/qa/README⟩.

[17] *GDB: The GNU Project Debugger*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨https://www.sourceware.org/gdb/⟩.

[18] *dbpmda(1)*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨https://man7.org/linux/man-pages/man1/dbpmda.1.html⟩.

[19] *Valgrind: About*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨https://valgrind.org/info/about.html⟩.

[20] *nc(1) - Linux man page*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨https://linux.die.net/man/1/nc⟩.

[21] *socket — Low-level networking interface — Python 3.11.1 documentation*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨https://docs.python.org/3/library/socket.html⟩.

[22] *pcp/INSTALL.md at main*. [online]. [cit. 2021-7-6].
Dostupný z: ⟨https://github.com/performancecopilot/pcp/blob/main/INSTALL.md⟩.

[23] *Ubuntu - Wikipedia*. [online]. [cit. 2022-12-31].
Dostupný z: ⟨https://en.wikipedia.org/wiki/Ubuntu⟩.

[24] *Qt4Removal - Debian Wiki*. [online]. [cit. 2022-12-31].
Dostupný z: ⟨https://wiki.debian.org/Qt4Removal⟩.

[25] *StatsD TCP Admin Interface*. [online]. [cit. 2022-12-20].
Dostupný z: ⟨https://github.com/statsd/statsd/blob/master/docs/admin_interface.md⟩.

[26] *pmstore(1) — Linux manual page*. [online]. [cit. 2022-12-20].
Dostupný z: ⟨https://man7.org/linux/man-pages/man1/pmstore.1.html⟩.

[27] *PCP Documentation - 4.5. The pmstore Command*. [online]. [cit. 2022-12-20].
Dostupný z: ⟨https://pcp.readthedocs.io/en/5.2.4/UAG/MonitoringSystemPerformance.html?highlight=pmstore#the-pmstore-command⟩.

[28] *pcp/pmda.h at 795222063a1d1341ef084412e9f9adefc4a84533 · performancecopilot/pcp*. [online]. [cit. 2022-12-20].
Dostupný z: ⟨https://github.com/performancecopilot/pcp/blob/795222063a1d1341ef084412e9f9adefc4a84533/src/include/pcp/pmda.h#L205⟩.

[29] *pmlogger(1) — Linux manual page*. [online]. [cit. 2022-12-20].
Dostupný z: ⟨https://man7.org/linux/man-pages/man1/pmlogger.1.html⟩.

[30] *statsd/metric_types.md at master · statsd/statsd*. [online]. [cit. 2022-12-21].
Dostupný z: ⟨https://github.com/statsd/statsd/blob/master/docs/metric_types.md#sets⟩.

[31] *DogStatsD*. [online]. [cit. 2022-12-20].
Dostupný z: ⟨https://docs.datadoghq.com/developers/dogstatsd/datagram_shell⟩.

[32] *Performance Co-Pilot — pcp 6.0.2-1 documentation*. [online]. [cit. 2022-12-24].
Dostupný z: ⟨https://pcp.readthedocs.io/en/latest/index.html⟩.

[33] *PCP Documentation - 1.2.6.1. Performance Metrics Name Space Diagram*. [online]. [cit. 2022-12-20].
Dostupný z: ⟨https://pcp.readthedocs.io/en/5.2.4/UAG/IntroductionToPcp.html#performance-metrics-name-space-diagram⟩.

[34] *PCP Documentation - 1.2.6. Performance Metrics Name Space*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨https://pcp.readthedocs.io/en/5.2.4/UAG/IntroductionToPcp.html#performance-metrics-name-space⟩.

[35] *PCP Documenation - 1.1. PCP Architecture*. [online]. [cit. 2022-12-24].
Dostupný z: ⟨https://pcp.readthedocs.io/en/latest/PG/ProgrammingPcp.html#pcp-architecture⟩.

[36] *PCP Documentation - 1.3.4. Application and Agent Development*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨https://pcp.readthedocs.io/en/5.2.4/UAG/IntroductionToPcp.html#application-and-agent-development⟩.

[37] *PCP Documentation - 4. Monitoring System Performance*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨https://pcp.readthedocs.io/en/5.2.4/UAG/MonitoringSystemPerformance.html⟩.

[38] *Graphviz Documentation - Output Formats - SVG*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨https://graphviz.org/docs/outputs/svg/⟩.

[39] *HDH Histogram C implementation*. [online]. [cit. 2022-12-23].
Dostupný z: ⟨https://github.com/HdrHistogram/HdrHistogram_c⟩.