VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

DIAGNOSTIKA SYSTÉMŮ ZALOŽENÝCH NA GNU/LINUX

DIAGNOSTICS OF GNU/LINUX-BASED SYSTEMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MARTIN HOFBAUER**
AUTHOR

VEDOUCÍ PRÁCE — **doc. RNDr. PAVEL SMRŽ, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Hofbauer Martin**
Program: Informační technologie
Název: **Diagnostika systémů založených na GNU/Linux**
Diagnostics of GNU/Linux-Based Systems
Kategorie: Algoritmy a datové struktury

Zadání:

1. Seznamte se s principy fungování operačního systému Linux, linuxového jádra a s funkcionalitou jednotlivých modulů, relevantních pro zjišťování případných chyb
2. Nastudujte existující řešení diagnostických systémů
3. Navrhněte a implementujte modulární program, který bude schopný provést diagnostiku systému GNU/Linux, kde jednotlivé moduly diagnostikují určité části systému
4. Vyhodnoťte chování programu v reálném a simulovaném prostředí
5. Vytvořte stručný plakát prezentující práci, její cíle a výsledky

Literatura:

- dle dohody s vedoucím

Pro udělení zápočtu za první semestr je požadováno:

- Funkční prototyp

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Smrž Pavel, doc. RNDr., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 1. listopadu 2021

Abstrakt

Tato bakalářská práce se zabývá tvorbou odlehčeného programu, který sbírá a ukládá informace ze systému založeného na operačním systému GNU/Linux. Dokáže také informovat o chybách v systému i jeho neobvyklém chování. Program cílí na maximální spolehlivost, izolovanost jednotlivých částí a jednoduchost použití. V teoretické části práce jsou popsány nedostatky existujících řešení a zasazení do kontextu, v praktické části je následně popsána architektura programu a její implementace. Program byl úspěšně nasazen v zadávající společnosti BringAuto, s jejíž spoluprací byla práce vytvořena. Ukázalo se, že je schopný zaznamenávat informace ze systému bez jeho přílišného zatížení. V případě selhání části systému velmi zjednodušil hledání příčin.

Abstract

This bachelor thesis focuses on the creation of the light-weighted program, which collects and saves information from GNU/Linux-based systems. Designed and developed solution can also inform about errors in the operating system and its unusual behavior. The program aims at maximum reliability, isolation of individual parts, and ease of use. The theoretical part describes problems of already existing solutions and focuses on the system monitoring context. The practical part then describes program architecture and its implementation. The program was successfully deployed in the company BringAuto, which collaborated with the creation of this thesis. It turned out that the program is able to record information from the operating system without its big load. In case of partial system failure, the program helped with finding its causes.

Klíčová slova

Linux, GNU, logování, diagnostika, sběr informací, C++, CMake, Asio, Modulog

Keywords

Linux, GNU, logging, diagnostics, collecting informations, C++, CMake, Asio, Modulog

Citace

HOFBAUER, Martin. *Diagnostika systémů založených na GNU/Linux*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. RNDr. Pavel Smrž, Ph.D.

Diagnostika systémů založených na GNU/Linux

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. RNDr. Pavla Smrže, Ph.D. Další informace mi poskytla partnerská firma BringAuto. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Martin Hofbauer
9. května 2022

Poděkování

Děkuji vedoucímu bakalářské práce doc. RNDr. Pavlu Smrži, Ph.D. za rady, trpělivost a ochotu při konzultacích. Poděkování patří také firmě BringAuto za poskytnutí podpory a možnosti testování aplikace v reálném prostředí.

Obsah

Kapitola 1

Úvod

Představme si, že vlastníme větší množství zařízení určených k monitoringu, na kterých běží operační systém založený na GNU/Linux. O těchto zařízeních chceme efektivně zjišťovat informace, přičemž není žádané tímto příliš zatěžovat systém. Také je potřeba mít možnost rychlého nasazení na systém bez instalování větších závislostí potřebných k běhu aplikace. Právě návrhem aplikace pro monitoring systému se tato bakalářská práce zabývá.

Od výsledné aplikace se očekává splnění specifikací popsaných v kapitole číslo 3.1 a také dlouhodobý běh bez jakéhokoliv selhání, který je u této aplikace kritický.

Hlavním důvodem zpracování této bakalářské práce je nedostatečné množství nástrojů na sběr informací ze systému, které by byly zároveň modulární, univerzální, jednoduché na použití a odlehčené.

Existují samostatné nástroje se zaměřením na sledování jedné konkrétní skupiny metrik, jako je např. pro měření teploty nástroj `lm_sensors`. Zde je ale problém, že neexistuje sjednocené ukládání zaznamenaných informací při použití tohoto nástroje v kombinaci s jinými nástroji a následné zpracování informací nelze jednoduše automatizovat. Také je problémem časování jednotlivých záznamů, kdy je potřeba si vytvořit vlastní skript a v něm využívat časování pomocí nástroje Cron. Ale s rostoucím počtem využitých měřících nástrojů se tento postup stává těžce udržovatelný.

Druhým extrémem je využití těžkopádných programů typu Prometheus, Datadog apod., které jsou sice velmi propracované, ale ke zjištění pár informací ze systému je potřeba velkých znalostí a dlouhé konfigurace. Navíc tyto programy spíše slouží k monitorování serverů a síťové infrastruktury a nejsou příliš vhodné pro zařízení s omezeným výpočetním výkonem.

Chybí tedy řešení, které by se nacházelo mezi těmito póly. Tedy řešení, které by malé monitorovací aplikace se specifickým zaměřením sjednotilo a odstranilo komplexitu, kterou přináší velké.

Kapitola číslo 2 se zabývá rozborem problematiky, existujícími řešeními a zasazením do kontextu vytvářené aplikace do celého systému zpracování dat. Popisuje také technologii S.M.A.R.T. důležitou pro sledování stavu disku a křížový překlad, který je využit pro integraci aplikace na zařízení s malým výpočetním výkonem. Kapitola číslo 3 se zabývá návrhem systému, do kterého spadá jeho architektura, komunikační protokol a všechny ostatní části spojené s návrhem vyvíjené aplikace. V kapitole číslo 4 je popsána implementace aplikace a také vybrané technologie s důvody jejich výběru. Kapitola číslo 5 se zabývá testováním aplikace v reálném i simulovaném prostředí. V závěru jsou shrnuty veškeré výsledky a nastíněn budoucí vývoj aplikace.

Kapitola 2

Rozbor řešené problematiky

2.1 Monitor systému

Monitorem systému rozumíme program nebo hardware, který monitoruje různé aspekty systému a následně tyto informace určitým způsobem poskytuje uživateli. Vyskytují se dva způsoby monitoringu, a to softwarové a hardwarové. Softwarové jsou často zabudovány v operačním systému, ale slouží spíše k rychlému nahlédnutí na aktuální stav zařízení, než na podrobnou analýzu a dlouhodobý vývoj stavu. Je zde také možnost využití samostatného programu třetí strany, které sice bývají často placené, ale zato velmi dobře zpracované.

Hardwarové monitorovací prostředky mohou být buď zabudovány přímo v zařízení, nebo připojovány k zařízení externě. Následně mohou poskytovat informace uživateli různým způsobem. Často bývají součástí základní desky jako samostatný čip s přístupem přes I^2C nebo sběrnici ISA [16]. Tato zařízení následně mohou monitorovat stav hardwaru, jako je například teplota uvnitř zařízení, rychlost otáček větráku nebo napětí poskytované zdrojem.

Nevýhodou softwarového monitoringu je přidané zatížení systému, ale při vybrání vhodných technologií a správném návrhu systému může být toto zatížení zanedbatelné.

Další důležitou součástí monitoringu je způsob ukládání dat. Je možné si vybrat mezi lokálním úložištěm a úložištěm vzdáleným. Lokální má tu výhodu, že je nezávislé na připojení k internetu, což je užitečné při monitorování systémů bez připojení. Na druhou stranu je obtížnější uložené informace získat, protože je potřeba se k zařízení fyzicky dostat a data stáhnout.

Naopak při velké frekvenci ukládání informací poskytuje lokální úložiště mnohem vyšší rychlost zápisu než zasílání přes síť. Také informace o stavu připojení k síti se jinak než lokálně zaznamenat nedá, protože při výpadku připojení danou informaci nelze zaslat.

V neposlední řadě je důležitý způsob uložení dat. Mohou být uložena buď v souborovém systému jako textové soubory, nebo v rámci databáze. Databáze mají spoustu výhod, jako například dotazování nad daty, konkurentní přístup, slučování dat a podobně. Mají ale i některé nevýhody, jako je například potřeba je do systému nainstalovat. Souborový systém se nachází v téměř každém systému a práce s ním je velmi jednoduchá. Také prohlížení dat může být pohodlnější v adresářové struktuře než v databázových tabulkách. Při využití databáze je také nutné vybrat co nejvhodnější databázi. Nejvíce se nabízí tzv. „time series“ databáze, které poskytují optimalizované ukládání dat pořízených v určitém čase. Jako příklad lze uvést databázi InfluxDB[1].

[1] `https://www.influxdata.com/`

Při velkém množství zaznamenaných dat může být také nutno řešit agregaci. Jedná se o proces, kdy jsou zaznamenaná data vyjádřena určitým shrnutím pro pozdější statistickou analýzu. Výhodou je, že se tím velmi šetří místo na paměťových médiích a prostředky na transport dat. Ovšem nevýhodou je ztráta informace, která agregací vznikne [18].

Možností a způsobů monitoringu se nabízí velké množství, a proto je důležité se před rozhodováním zamyslet, jaké jsou požadavky a na jejich základě vybrat, jaká řešení budou použita.

2.2 Existující řešení

Pro monitoring systému a detekci chyb již aplikace existují, ale jak bylo zmíněno v úvodu, vyskytují se dva extrémy, a to buď příliš specifický program pro monitoring úzké skupiny metrik, nebo až příliš komplexní. Zároveň žádný neobsahuje veškeré požadavky specifikované v sekci číslo 3.1.

2.2.1 Prometheus

Jedná se o monitorovací nástroj s otevřeným kódem, který data sbírá dotazováním monitorovaných systémů (tzv. „scraping“). Na monitorovaných zařízeních běží služby (tzv. „exporters“), které vystaví monitorovaná data na koncový bod. Z toho jsou následně v předem nastavených časových intervalech přes protokol HTTP odebírána serverem. Pokud nastane určitá chyba, je zaslána serverem do komponenty zvané „Alertmanager“, která na problém může upozornit například přes Slack nebo email [20].

Co se týče vizualizace dat, tak Prometheus poskytuje pouze rozhraní na získávání dat a dotazovací jazyk PromQL. Pro vizualizaci je tedy nutné použít software 3. strany, jako např. Grafana [2]. Je to ale jedna z nevýhod, protože pro použití tohoto softwaru je nutné ho nejdříve nakonfigurovat, a to může být poměrně časově náročné.

Další nevýhodou tohoto nástroje je fakt, že je postavený na dotazování monitorovaných systémů serverem, a tudíž je komplikované vyřešit situaci, kdy je potřeba zaslat určitou informaci serveru po dokončení úlohy. Lze použít tzv. „Pushgateway“ [3], ale přidává se tím na komplikovanosti a dokumentace tuto praktiku příliš nedoporučuje.

2.2.2 Nagios

Nagios je monitorovací nástroj navržený pro běh na operačním systému založeném na Linuxu. Dokáže monitorovat ale i operační systémy Windows i UNIXové systémy. V pravidelných intervalech kontroluje aplikační, síťové a serverové zdroje. Může například monitorovat vytížení procesoru, počet běžících procesů, zaplnění paměti a podobně. Také dokáže sledovat služby jako SMTP, POP3, HTTP a jiné síťové protokoly. Tyto kontroly jsou nazývány aktivní a jsou prováděny ze strany Nagiosu. Existují také pasivní kontroly, které jsou prováděny z externích aplikací připojených k Nagiosu [23].

Lze využít buď verzi zdarma Nagios Core s otevřeným zdrojovým kódem, nebo placenou verzi Nagios XI, která nabízí velké množství vylepšení.

Výhodou tohoto softwaru je jeho obrovská konfigurovatelnost, která ale přidává velké množství složitosti a pro nezkušeného uživatele je velmi těžké vůbec zprovoznit a nakonfigurovat základní logovací aplikaci.

[2] `https://grafana.com/`

[3] `https://prometheus.io/docs/practices/pushing/`

2.2.3 Datadog

Jedná se o monitorovací a analytické řešení pro IT oblast a DevOps[4], které může být použito k pozorování výkonnostních metrik a také ke sledování událostí v infrastruktuře a cloudových službách[5]. Může sledovat služby jako např. databáze, servery a jiné nástroje.

Lze využít buď jako varianta „Software as a Service“ (SaaS), nebo „On-Premise“ software. V prvním případě je uživateli poskytnuto kompletní řešení hostování aplikace třetí stranou a uživatel pouze platí na základě předplatného. U druhé varianty uživatel hostuje aplikaci na vlastních serverech, kde je výhodou kompletní kontrola, ale nevýhodou je velká cena hardwaru a neustálá údržba.

Datadog je podporován na operačních systémech Windows, Linux a Mac a jako poskytovatel SaaS může být využit Microsoft Azure, AWS nebo Google Cloud Platform [19].

Podobně jako u ostatních existujících řešení, i zde existují samostatné programy, které se starají o sbírání informací. Konkrétně se jim říká agenti. Ti musí být nainstalováni v systému pro to, aby poté bylo možné vytvářet vlastní programy pro sběr specifických informací. U systému Datadog se tomu říká „Custom Agent Check“ a jeho tvorba není příliš složitá. Je potřeba pouze mít již nainstalovaného agenta, vytvořit konfigurační soubor pro agenta a nakonec vytvořit skript v jazyce Python, který bude sbírat a vystavovat informace po následujícím restartu agenta. Nevýhodou je, že se pro tyto sběry využívá Python, který patří do skupiny interpretovaných programovacích jazyků a programy v něm jsou pomalejší, než jiné napsané například v C++, tudíž náročnější sběry informací mohou velmi zatížit systém.

Další nevýhodou je fakt, že Datadog agenti nejsou příliš odlehčení a mají poměrně velké množství závislostí, mezi kterými je například interpret programovacího jazyka Python, nebo dokonce i Docker.

2.2.4 Munin

Tento nástroj s otevřeným zdrojovým kódem slouží k dlouhodobému sledování informací o systému a jejich zobrazování ve formě grafů. Po jejich intuitivním zobrazení je možné jednoduše odhadnout aktuální stav zařízení. Oproti ostatním monitorovacím systémům je velmi odlehčený a jednoduchý na použití i pro uživatele, který nikdy podobný program nepoužil. Lze u něj také nastavovat hraniční hodnoty, po jejichž přesažení bude uživatel informován formou mailu nebo jiným způsobem.

Skládá se ze ze dvou částí a to tzv. „master“ a „Munin nodes“, kterým budeme říkat uzly. Master se stará o sbírání dat od uzlů v pravidelných, pěti minutových intervalech, které jsou naplánovány pomocí softwarového démona Cron[6] [14]. V těchto intervalech vždy master spustí zásuvné moduly, které jsou přiřazeny k jednotlivým uzlům. Zásuvný modul je jakýkoliv spustitelný soubor, který je kompatibilní s komunikačním protokolem Munin, sbírá informace o systému a následně je vystavuje na standardní výstup. Master tyto informace následně sesbírá a uloží.

Pro Munin existuje velké množství zásuvných modulů, ale je velmi jednoduché vytvořit si vlastní v libovolném programovacím jazyku Pokud je zásuvný modul spuštěn s argumentem `config`, tak je očekáváno, že vypíše na standardní výstup metadata popisující graf, ve kterém budou sledované hodnoty uloženy. V případě, že je spuštěn bez jakéhokoliv

[4]Spojení slov „development“ a „operations“, slouží k zajištění průběžného doručování kvalitních produktů a služeb zákazníkům.

[5]Internetová služba dostupná odkudkoliv.

[6]Jedná se o standardní UNIXový nástroj používaný právě k pravidelnému spouštění programů.

argumentu je očekáváno, že vypíše na standardní výstup již sledovaná data ve formátu klíč-hodnota [15].

Program Munin je nejblíže k požadovanému řešení, ale bohužel nesplňuje všechny požadavky. Jeho hlavní nevýhodou je monitorování v pevně daných pětiminutových intervalech, což není příliš užitečné v momentě, kdy je potřeba získat informaci ze systému například jednou za týden. Také mu chybí možnost okamžitého zaznamenání informace při vzniku určité události. Jeho další nevýhodou je také to, že je příliš orientovaný na grafy a nenabízí příliš pohodlnou práci s čistými uloženými daty.

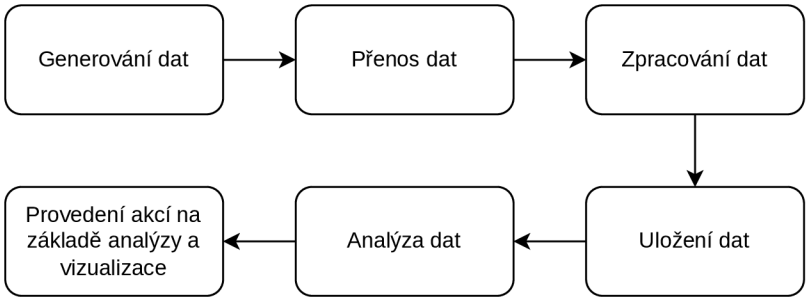
2.2.5 Shrnutí

Existuje velké množství monitorovacích systémů pro operační systém Linux, ale žádný z nich nesplňuje všechny požadavky, které jsou v sekci číslo 3.1 definovány. Téměř všechny jsou vytvořeny hlavně pro sledování serverů a síťové infrastruktury, a proto nejsou příliš vhodné pro monitoring zařízení s omezenými výpočetními zdroji, kde některé nemají ani přístup k internetu.

Také mají všechna řešení jednu společnou vlastnost, a to tu, že trvá poměrně dlouho, než se uživatel naučí celý princip fungování a je schopný řešení nasadit pro vlastní účely. K téměř všem existují knihy, které popisují jejich využití, což na jednu stranu poukazuje na jejich široké využití, ale také na jejich komplexitu a složité použití.

2.3 Průběh zpracovávání dat

Aplikace Modulog bude generovat množství dat na různých zařízeních. Tato zařízení nemusí být pouze autonomní vozidla společnosti BringAuto, ale může se jednat i o různé vestavěné systémy nebo i servery. Všechna generovaná data musí být zpracována a ve většině případů i analyzována. Jejich cestu od vzniku až po analýzu lze vidět na obrázku číslo 2.1. Zde se jedná pouze o jednu možnost z mnoha, pořadí se může lišit podle toho, jestli se jedná o proudově nebo blokově orientované zpracování dat (viz sekce číslo 2.3.4), na záměru práce s daty apod. Dále také některé technologie slučují vícero fází do jedné, nebo i fáze přidávají [12].



Obrázek 2.1: Průběh cesty dat

2.3.1 Generování dat

Zde se jedná o samotné vytvoření informace, jako je v programu Modulog záznam jakékoliv informace ze systému. Obecně se může jednat i o zaznamenání akce uživatele na webové stránce a podobně. Data jsou zatím nestrukturovaná a jedná se pouze o čistou informaci.

2.3.2 Přenos dat

V této fázi jsou již data připravena na přenos ze zařízení na server. To může probíhat buď okamžitě, pokud je informace důležitá a dále bude muset být co nejdříve zpracována, nebo může být zaslána se zpožděním v dávkách s dalšími informacemi. Také je možná kombinace, kdy důležité informace budou zasílány okamžitě a méně důležité v dávkách.

Podle situace je možné využít různé technologie. Pro vestavěné systémy lze použít protokol MQTT, který poskytuje možnost odesílat data z velkého množství zařízení zaráz, a také je vhodný do momentů, kdy zařízení nemají stabilní internet. Tento protokol je standardizovaný a velmi odlehčený, takže je vhodný pro zařízení bez příliš velkých výpočetních možností.

Pro přenos dat lze použít i projekt Apache Kafka, který má otevřený zdrojový kód a vznikal ve společnosti LinkedIn. Jedná se o platformu určenou ke streamingu událostí, což je činnost zachytávání událostí ze zdrojů, jako jsou různé senzory, zařízení a podobně.

Kafka kombinuje tyto tři schopnosti [5]:

- Číst a zapisovat toky událostí včetně importování či exportování dat z jiných systémů
- Ukládání toků událostí dlouhodobě a spolehlivě na dobu neurčitou
- Zpracování toků událostí v reálném čase nebo i zpětně

Projekt Kafka má ovšem i určité nevýhody pro přímé napojení na koncová zařízení v IoT [4]:

- Není připravený na obrovské množství připojených zařízení
- Kafka klienti jsou poměrně složití a spotřebují více výpočetního výkonu, než by bylo vhodno
- Je potřeba stabilního připojení
- Chybí mu určité IoT možnosti jako „last will and testament“[7]

Z těchto důvodů nebývá napojený přímo na koncová zařízení, ale často se využívá v kombinaci s protokolem MQTT, který je na tyto problémy připravený.

2.3.3 Uložení dat

Tato fáze opět může být brána různými způsoby. V některých situacích nemusí být vůbec využita, kdy je pouze potřeba kontrolovat příchozí data v reálném čase a na základě toho vykonávat akce bez potřeby jejich skladování. Pro ukládání velkého množství dat lze použít například databáze jako InfluxDB, Cassandra nebo PostgreSQL.

2.3.4 Zpracování dat

Zpracování dat lze rozdělit na 2 hlavní způsoby: proudově a blokově orientované [2].

[7]Klient předem specifikuje, co má být provedeno po tom, co je odpojen od internetu.

Proudově orientované zpracování

Data jsou v tomto případě zpracována co nejdříve to jde a uložena jsou až po zpracování. V některých případech nemusí být ani uložena, pokud jsou příliš velká nebo po zpracování nepoužitelná. Tento přístup se využívá v systémech, kde musí být co nejnižší odezva mezi příchodem dat a jejich efektem, což může být například provedení akce nebo notifikace uživatele. Je zde potřeba také co největší dostupnosti, proto se využívají tyto techniky:

- Distribuce zpracování mezi více zařízení
- Kopírování informace, aby při poruše jednoho zařízení mohla být zpracována na jiném
- Pararelní zpracování
- Zpracování ve vnitřní paměti pro co nejmenší odezvu

Blokově orientované zpracování

U tohoto způsobu zpracování dat není příliš kritická jeho odezva. Data se totiž zpracovávají po blocích a tudíž je nutné čekat, než je připravené určité množství dat, které se až poté může zpracovat. Tuto metodu lze použít pro získání informací na základě většího počtu dat, jako je například střední hodnota či hledání extrémů za určitou dobu.

Opět existuje velké množství technologií pro zpracování dat, kde každá má své výhody i nevýhody. Projekt Kapacitor lze využít jak pro proudově orientované zpracování, tak i pro blokově orientované. Dále lze do této kategorie zařadit projekty Apache Spark, nebo Apache Hadoop.

2.3.5 Vizualizace dat

Tato fáze slouží k přehlednému zobrazení informace uživateli. Může se jednat o jednoduchou webovou stránku zobrazující pár hodnot, nebo i velké množství komplexních grafů a nasbíraných informací. V některých případech může být tato fáze i přeskočena, pokud je celý proces čistě automatizovaný a na základě zpracovaných informací jsou samy prováděny určité akce.

Pro vizualizaci lze využít projekt s otevřeným zdrojovým kódem napsaný v jazycích TypeScript a Go pojmenovaný Grafana. Je vysoce konfigurovatelný a nabízí využití přípojných modulů pro rozšíření její funkcionality. Na obrázku číslo 2.2 lze vidět, jak může vypadat vizualizace pomocí projektu Grafana.

Obrázek 2.2: Vizualizace pomocí projektu Grafana [7]

2.4 Technologie S.M.A.R.T.

Jednou z věcí, které je potřeba monitorovat u autonomního vozidla, je právě stav disku, na kterém je uložený celý systém sloužící k běhu vozidla. Ovšem jako všechny elektronické součástky, tak i v disku může dojít k poruše. Ty se dělí na 2 větve: předvídatelné a nepředvídatelné.

Nepředvídatelné vznikají náhlým porušením materiálu, jako např. poškození určité součástky pádem a následným otřesem paměťového média. Tyto poruchy se nedají předvídat a dá se jim částečně zabránit umístěním média na dobře chráněné místo, na kterém nejsou vibrace a podobně.

Do předvídatelných patří poruchy, které vznikají s postupem času (jako např. postupná degradace materiálu). Některé z těchto poruch lze částečně předvídat a brzkým zásahem zachránit data před jejich ztrátou. A právě k tomuto předvídání může pomoci technologie S.M.A.R.T., kterou lze použít na pravidelné kontroly disku.

Pro využití této technologie je nutné, aby monitorované médium tuto technologii podporovalo. Může se vyskytovat jak na pevných discích, tak i na SSD či micro SD kartách.

Velká část této technologie je standardizovaná, jako například způsob práce s daty nebo práce s chybovými zprávami. Ovšem některé části jsou jen z části standardizovány a každý výrobce disků se k nim může postavit jinak. Do této skupiny patří například S.M.A.R.T. atributy, které jsou základním stavebním kamenem celého procesu zachytávání informací o disku. Určitým způsobem popisují části zařízení, ze kterých lze odvodit aktuální stav disku. Můžeme si je představit jako například teplotu zařízení nebo počet zapsaných bajtů. Jeden atribut musí obsahovat:

- Jméno
- Hodnotu, která je individuálně definována výrobcem
- Normalizovanou hodnotu - číslo mezi 1 a 253, kde nižší číslo znamená horší stav
- Hranici - číslo, pod které když klesne normalizovaná hodnota, tak je nastaven S.M.A.R.T. status jako chybový

S.M.A.R.T. status je informace nabývající dvou hodnot: hranice přesažena a hranice nepřesažena. Pokud byla přesažena, tak je pravděpodobné, že brzy na disku nastane porucha. Ovšem nemusí se tak dít, může pouze dojít ke snížení výkonu, nebo nemusí nastat vůbec žádný problém. Pokud hranice nebyla přesažena, tak to neznamená, že je disk v naprostém pořádku, ale dá se očekávat, že pravděpodobnost poruchy není příliš vysoká. Můžeme tedy vidět, že S.M.A.R.T. status spíše slouží k upoutání pozornosti na disk a je nutno dále provést akce jako záloha dat a bližší prozkoumání problému [17].

Pro získání S.M.A.R.T. informací lze použít set programů `smartmontools`, který je dostupný pro velké množství operačních systémů. Nativně podporuje pouze ovládání přes příkazovou řádku, ale lze stáhnout rozšíření třetích stran, která přidávají možnost práce s GUI [8]. `Smartmontools` podporuje výstup ve formátu json, a proto je jednoduché automatizované vyhodnocování dat vytvořených tímto setem programů.

2.5 Křížový překlad (cross compilation)

Křížový překlad je technika určená k překladu zdrojového kódu jedné architektury do architektury jiné. Například pokud bychom chtěli přeložit aplikaci určenou pro mobilní telefon z počítače. V některých případech se bez této techniky nelze obejít a v jiných ušetří spoustu času:

- Při kompilaci pro vestavěné systémy, kdy cílové zařízení nemá dostatečný výkon, je nutné křížový překlad použít. V opačném případě by překlad trval velmi dlouho, nebo by nebylo možné ho vůbec provést.
- Při kompilaci aplikace pro více systémů - pokud je potřeba podporovat větší množství systémů, na kterých aplikace poběží, tak je časově a cenově velmi nevýhodné aplikaci zkompilovat na každém systému zvlášť. Proto se využije křížový překlad a vše se provede z jednoho počítače, kde budou nastaveny pouze prostředí pro všechny cílové systémy.
- Dlouhé kompilace - pokud je potřeba zkompilovat program, který je rozsáhlý a čas kompilace by byl velký, tak je možné použít křížovou kompilaci a překlad provádět na více různých zařízeních zaráz.

V našem případě bude potřeba křížový překlad využít, protože cílovou aplikaci bude třeba podporovat na zařízeních s různými architekturami a zároveň sestavování aplikace bude automatizováno, takže více kompilací pro různé architektury bude prováděno z jednoho serveru. Více o praktické stránce naleznete v kapitole číslo 5.2

[8] `https://gsmartcontrol.shaduri.dev/downloads`

Kapitola 3

Návrh systému

3.1 Specifikace požadavků

Spolupracující firma požaduje software na sledování stavu hardwarových komponent umístěných na autonomních vozidlech. V budoucnu jich bude vlastnit vícero a na těchto vozidlech bude potřeba zjišťovat různé parametry, jako například životnost disků, vytížení procesoru a jiné potřebné informace. Ale také bude třeba sbírat tyto informace z jednotlivých senzorů na vozidle a nadále s nimi pracovat - uložit lokálně na zařízení, nebo posílat na vzdálený server. Také společnost potřebuje, aby nasazení aplikace na vozidlo bylo velmi jednoduché bez pročítání dlouhé dokumentace. V neposlední řadě se od aplikace očekává, aby vozidlo příliš neochuzovala o výpočetní výkon. Taktéž by firma preferovala vlastní nebo „open-source“ řešení, aby nebyla závislá na již existujících proprietárních.

Vytvořený program se bude jmenovat Modulog a bude dostupný na platformě GitHub.

3.1.1 Definice aplikace Modulog

Je potřeba, aby byla aplikace rozdělena na dvě části - jedna, která se stará o ukládání veškerých informací ze systému a druhá, která se skládá ze samostatných programů sbírajících tyto informace. Programy z druhé části se nazývají agenti. Po spuštění by byly sbírány informace buď v nekonečné smyčce, nebo pouze určitou dobu na základě nastavení jednotlivých agentů.

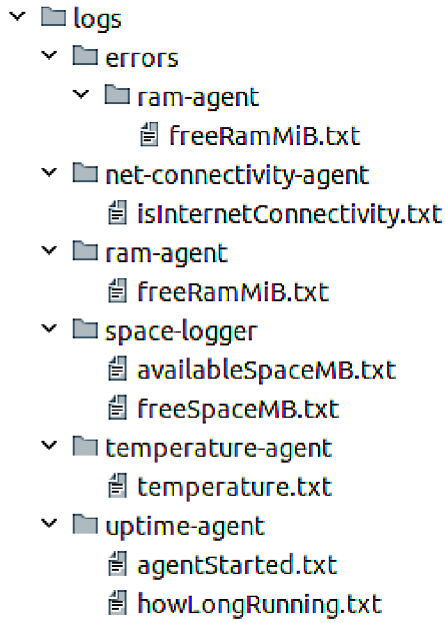
Aplikaci Modulog by bylo možné před spuštěním nakonfigurovat a vybrat si, jestli se informace budou ukládat buď do složkové struktury, nebo do jednoho souboru. V případě složkové struktury by každá složka představovala jednoho agenta a v ní by se poté nacházely jednotlivé soubory, kde by název souboru byl klíč informace a v něm by byl seznam hodnot přiřazených k tomuto klíči. Pro lepší představu je struktura uvedena na obrázku číslo 3.1.

Agenti musí mít možnost zaznamenat jak klasickou informaci ze systému, tak také chybové informace. Pokud je tedy zaznamenána chyba, aplikace Modulog musí na tuto chybu upozornit v samostatné složce `errors`, protože při větším počtu agentů je těžké procházet všechny ostatní složky, ale tímto způsobem se stačí podívat pouze do jedné a uživatel okamžitě vidí, v jakém agentovi nastala jaká chyba. Z obrázku číslo 3.1 lze tedy vyčíst, že v agentovi `ram-agent` došlo k chybě u klíče `freeRamMiB`. Obsah souboru `freeRamMiB.txt`, který je přiřazen klíči `freeRamMiB` vypadá následovně:

```
[2022-01-22T13:52:37] 5358 LOG
[2022-01-22T13:52:38] 5342 LOG
```

```
[2022-01-22T13:52:39] 5331 ERROR
[2022-01-22T13:52:40] 5341 LOG
[2022-01-22T13:52:41] 5331 ERROR
[2022-01-22T13:52:42] 5350 LOG
```

Lze tedy vidět, že byla dvakrát překročena předem definovaná hranice dostupné RAM.



Obrázek 3.1: Složková struktura zaznamenaných informací

Aplikace Modulog také musí zaznamenat, když u některého agenta nastane chyba a neukončí se podle definovaného komunikačního protokolu. Seznam těchto agentů musí ukládat do souboru `crashedAgents.txt` s časovým razítkem ukončení agenta. Také musí kontrolovat, jestli agenti fungují správně a neuváznuli při vykonávání činnosti. Z toho důvodu jim bude v pravidelných intervalech posílána zpráva `IS_ALIVE`, na kterou musí odpovědět.

Další velmi důležitou vlastností programu Modulog musí být tvorba vlastních agentů. K tomu bude potřeba vytvořit klienta, který by se integroval do nových agentů a pomocí něho by se jednoduše daly zasílat sesbírané informace do hlavní části programu. Také je nutné, aby uživatel mohl mít možnost vytvořit sdílený soubor, který by se po spuštění programu Modulog rozdistribuoval všem agentům - v něm by mohly být sdílené informace pro všechny agenty, jako například jak často zaznamenat informaci a podobně.

V případě nevyužití sdílené konfigurace by se očekávalo, že agent bude nastaven individuálním konfiguračním souborem. Tato praktita již nespadá pod program Modulog, protože už záleží na tvůrci agenta, jak se k této problematice postaví. Agenti vytvořeni pro Modulog budou ale využívat konfigurační soubor ve formátu JSON, který může pro agenta monitorujícího volnou RAM vypadat následovně:

```
{
        "id": "ram-agent",
        "freeNotSmallerThanMiB": 5340,
        "logIntervalSec": 1
}
```

Zde nastavíme jméno agenta na „ram-agent“, dále hranici volné paměti na 5340 MiB a interval zaznamenání informace na jednu vteřinu.

Jedinou povinnou informací, kterou agent bude poskytovat při vytváření spojení s aplikací Modulog je právě jeho jméno. To bude využito jako identifikace nově vytvořené složky pro ukládání informací. V případě více agentů se stejným jménem jsou jejich zaznamenané informace sloučeny. Toto jméno ale nemusí být součástí žádného konfiguračního souboru, stačí, když bude uloženo v proměnné ve zdrojovém kódu agenta.

3.2 Konceptuální rozdělení

Aplikace bude rozdělená do dvou konceptuálních celků - agenti, kteří se budou starat o samostatné sbírání informací ze systému a jádrem, kterému budou tyto informace zasílány a následně je bude zpracovávat. Důležité je, aby selhání agenta neovlivnilo zbytek aplikace, takže jako nejlepší řešení se nabízí, že agenti budou samostatně běžící procesy, které svým selháním nemůžou ukončit celou aplikaci.

Systém tedy bude rozdělen do tří knihoven: Core, Communication, AgentClient.

3.2.1 Knihovna Core

Tato knihovna bude jádrem celého programu. Na začátku vykonávání se postará o zpracování souboru, ve kterém jsou poznačeni požadovaní agenti. Následně tyto agenty postupně začne jako samostatné procesy pouštět, po každém spuštění s nimi naváže spojení, vymění si potřebné informace a následně začne přijímat zaznamenané informace. Pokud během vytváření alespoň u jednoho agenta nastane chyba a core s ním nenaváže spojení, tak je celý program ukončen.

Core poté přijímá zasílané informace agentů do té doby, dokud program není přerušen signálem, nebo dokud nejsou všichni agenti ukončeni. V případě přerušení musí všechny agenty ukončit a následně ukončit i sebe.

3.2.2 Knihovna Communication

Knihovna sloužící ke komunikaci mezi agenty a core. Abstrahuje zasílání a přijímání zpráv a v případě změny komunikační technologie stačí nahradit pouze tuto knihovnu bez zásahů do zbytku architektury.

Při přijmutí zprávy upozorní čekatele na příchozí zprávu. Stará se také o spojení dlouhých zpráv zaslaných ve více částech a jejich následné poskytnutí.

3.2.3 Knihovna AgentClient

Tato knihovna je určena k jednoduché implementaci nových agentů. Pokud je potřeba vytvořit vlastního agenta se specifickým chováním, tak stačí pouze do projektu přidat tuto knihovnu, poté ji inicializovat pomocí funkce `initClient()` a následně je možné posílat informace pomocí funkce `sendLog()`. O všechno ostatní se stará knihovna sama, jako např. inicializace spojení a odpovídání na kontrolní zprávy.

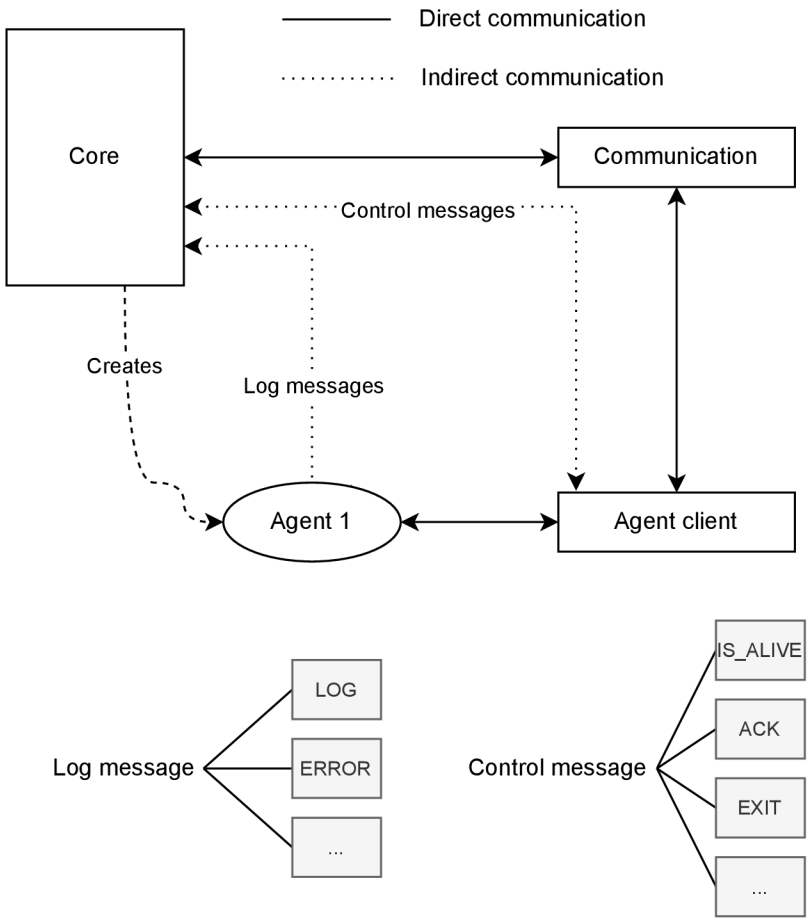
Na obrázku 3.2 můžeme vidět, jak spolu knihovny navzájem komunikují.

V systému existují 2 typy zpráv: jedny obsahující informaci k uložení (tzv. „log message“), druhé sloužící ke komunikaci mezi jednotlivými celky (tzv. „control message“).

U prvního existuje několik typů zprávy, jako např. jestli se jedná o informační, chybovou nebo jinou zprávu.

U druhého typu zprávy je nutno specifikovat sémantiku zprávy, zde je neúplný seznam pro lepší představení:

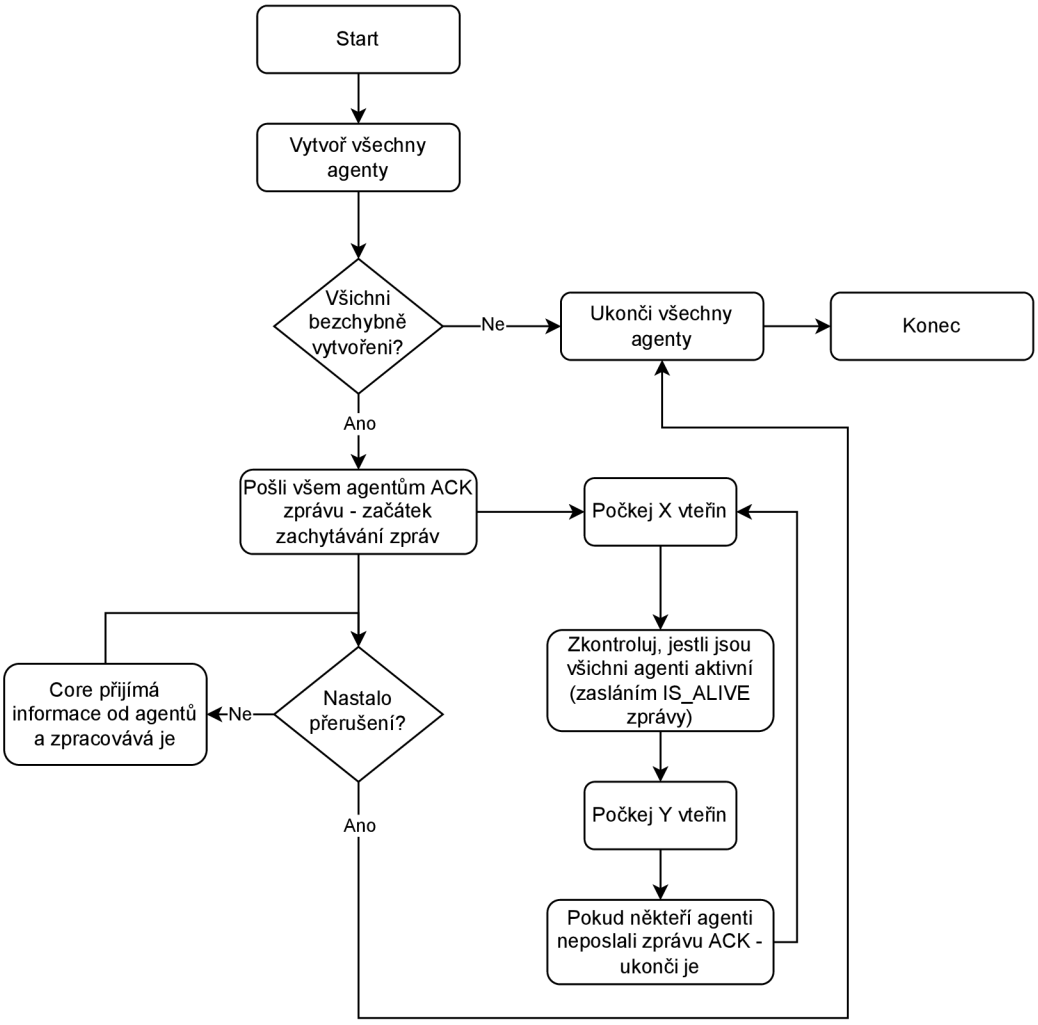
- `IS_ALIVE` - zpráva je zasílána agentům za účelem zjistit, jestli jsou aktivní. Pokud na tuto zprávu neodpoví, tak jsou poté ukončeni
- `ACK` - agenti odpovídají touto zprávou na zprávu `IS_ALIVE`
- `EXIT` - zprávu posílá agent a oznamuje tím, že plánuje být za chvíli ukončen (pokud například již zaznamenal vše, co potřeboval)



Obrázek 3.2: Architektura systému

3.3 Průběh programu

Na obrázku číslo 3.3 lze vidět vývojový diagram chování systému, který představuje průběh programu od spuštění, až po jeho ukončení.



Obrázek 3.3: Vývojový diagram chování systému

3.4 Komunikační protokol

3.4.1 Vytvoření agenta

Core vytvoří agenta, kterému následně zašle konfigurační soubor - tento soubor se zvolí před začátkem běhu programu a je zaslán všem agentům. Lze jej použít jako sdílenou konfiguraci pro všechny agenty. Agent následně odpoví zprávou ACK, do které vloží své jméno - core bude pod tímto jménem následně ukládat všechny informace od tohoto agenta.

3.4.2 Agent se chce ukončit

Agent zašle kontrolní zprávu typu EXIT do core. Core pošle zpět zprávu EXIT. Agent se nyní může ukončit. Pokud po několika vteřinách není agent ukončen (tento čas lze nastavit v konfiguračním souboru programu), tak core zašle signál SIGKILL a agenta tak ukončí.

3.4.3 Core ukončí agenta

Core zašle kontrolní zprávu typu EXIT agentovi. Následně čeká několik vteřin (čas nastavitelný v konfiguračním souboru programu) a poté agentovi zašle signál typu SIGKILL.

Kapitola 4

Implementace

4.1 Výběr technologií

4.1.1 Programovací jazyk

Jednou z nejdůležitějších částí při výběru použitých technologií je právě programovací jazyk. V tomto případě je nutné zvolit takový, pomocí kterého bude možné napsat rychlou a efektivní aplikaci. Také je důležité volit jazyk kompilovaný, aby na cílových zařízeních nemusel být přítomný interpret (kvůli zpomalenému vykonávání kódu a také rychlosti nasazování aplikace bez nutnosti instalovat dodatečné programy). Po zvážení všech vyjmenovaných požadavků byl vybrán programovací jazyk C++. Mohl by se také použít programovací jazyk C, ale C++ nabízí objektové programování a v kombinaci se systémem CMake je možné vytvářet čitelný kód s jednoduchou správou knihoven třetích stran [1].

4.1.2 Izolace agentů

V aplikaci se budou vyskytovat agenti, kteří poběží ve smyčce a budou zaznamenávat určité informace o systému. Tito agenti budou spuštěni na začátku běhu programu a ukončí se buď sami, nebo po přerušení hlavní aplikace signálem. Agenti mohou běžet dvěma způsoby: buď jako součást nových vláken nebo samostatných procesů [22].

Pokud by agenti byli vytvářeni pomocí vláken, tak by se velmi pomohlo jednoduššímu vývoji, jako je například pohodlnější hledání chyb v jednotlivých agentech debuggerem. Na druhou stranu, vlákna sdílí určité systémové zdroje, takže se nejedná o úplnou izolovanost. Pokud v C++ například vytvoříme nové vlákno, ve kterém nastane porušení ochrany paměti, tak se okamžitě ukončí celá aplikace. A právě proto je kompletní izolovanost v našem případě kritická, protože není žádané, aby selhání jednoho agenta ukončilo celou aplikaci. Kvůli tomu tedy bylo vybráno, že agenti poběží v samostatných procesech a ne vláknech. Tato skutečnost přidá i drobnou výhodu, že se na agenty můžeme dívat jako na samostatné programy, které mohou být vyvíjeny úplně samostatně.

4.1.3 Použité knihovny

Správa procesů

V aplikaci se nachází agenti, kteří zaznamenávají různé informace o systému (viz 3). Potřebujeme, aby běželi nezávisle na zbytku aplikace a svým chováním ji nemohli nikterak ovlivnit. Proto poběží jako samostatné procesy a ne jako samostatná vlákna z důvodu větší

izolovanosti. Pro vytváření těchto procesů lze použít volání funkce `fork()` programovacího jazyka C, ale pro čitelnější kód byla zvolena knihovna třetí strany, která by umožňovala procesy jednoduše vytvářet i ukončovat. Na následujících řádcích se nachází knihovny, mezi kterými bylo rozhodováno a jejich plusy i mínusy:

- **sheredom/subprocess.h** [1] - Tato knihovna vypadala zpočátku velmi užitečně. Je průběžně aktualizována a jedná se o tzv. „header-only" knihovnu - všechny definice funkcí, tříd a maker jsou uloženy v jediném hlavičkovém souboru a tím pádem není nutné knihovnu separátně kompilovat a instalovat. Po integraci ale bylo zjištěno, že nedokáže správně upozorňovat na chyby při spouštění neexistujících spustitelných souborů, tudíž byla při výběru zavrhnuta - v budoucnu je možné na ni přejít, pokud bude chyba opravena.
- **eidheim/tiny-process-library** [2] - Knihovna není příliš aktualizována, tak byla zvážena jiná.
- **Boost.Process** [3], **POCO** [4] - Tyto knihovny by určitě postačily, ale bohužel se jedná o příliš velké systémy, které jsou zbytečně rozsáhlé pro pouhou správu procesů.
- **DaanDeMeyer/reproc** [5] - Tato knihovna byla nakonec použita, je aktivně vyvíjena, její použití je velmi jednoduché a v případě pro aplikaci Modulog prozatím dostačuje.

Správa procesů bude napojena na jednoduché rozhraní, takže i kdyby v budoucnu byla potřeba změnit knihovnu na správu procesů, tak se nejedná o příliš velký problém.

Práce s formátem JSON

JSON je formát, který se používá k uchovávání a přenosu dat. Je velmi odlehčený a oproti například XML se s ním velmi pohodlně pracuje díky jednoduché syntaxi.

Lze ho použít jako formát pro přenos zpráv nebo jako formát pro konfigurační soubory aplikace. Pro oba tyto případy bude také využit, a proto je potřeba vybrat knihovnu pro C++, která usnadní práci při zpracovávání tohoto formátu. Pro C++ existuje velké množství knihoven, dokonce lze najít i srovnání jak z rychlostního, tak i z paměťového hlediska [24].

Na základě těchto statistik je nejefektivnější rapidJSON [6]. Ovšem efektivita hraje důležitou roli pouze při serializaci zpráv, která je při velké frekvenci zasílání informací velmi kritická, ale v tomto případě bude JSON použit pro serializaci zpráv pouze dočasně a při optimalizaci aplikace bude použita pro serializaci technologie na způsob Google Protocol Buffers [7].

Z těchto důvodů je potřeba pouze knihovna, která bude jednoduše integrovatelná a lehká na použití. Proto byla vybrána velmi populární nlohmann/json[8], u které stačí pouze nakopírovat hlavičkový soubor do projektu a je připravena k použití. Její použití je následně velmi čitelné, k získání hodnoty klíče `timeout` ze souboru `config.json` stačí pouze následující řádky:

[1] `https://github.com/sheredom/subprocess.h`
[2] `https://github.com/eidheim/tiny-process-library`
[3] `https://www.boost.org/`
[4] `https://docs.pocoproject.org/`
[5] `https://github.com/DaanDeMeyer/reproc`
[6] `https://github.com/Tencent/rapidjson`
[7] `https://github.com/protocolbuffers/protobuf`
[8] `https://github.com/nlohmann/json`

```
1     std::ifstream ifs("config.json");
2     json config = json::parse(ifs);
3     int timeout = config["timeout"];
```

Zpracování argumentů

Zde se nabízí využití knihovny jazyka C `getopt()`, která nenabízí příliš propracované zpracování argumentů, ale na druhou stranu je součástí specifikace POSIX. Dále je možné použít knihovnu Boost.Program_options[9], kvůli které je ale zbytečné do celého projektu integrovat Boost, takže byla zavrhnuta. Další možnou variantou je knihovna cxxopts[10], která podporuje formát argumentů GNU standardu. Její integrace je velmi jednoduchá, stačí pouze nakopírovat hlavičkový soubor do projektu a poté je připravena k použití. Jelikož je tato knihovna velmi dobře zpracována a nemá příliš velkou konkurenci, tak byla použita.

Ukládání záznamů

Při zaznamenání informace o systému je nutno ji někam uložit. Modulog bude podporovat dva způsoby, a to buď ukládání do složkové struktury, nebo vše do jednoho souboru. K tomu bude využita knihovna partnerské společnosti s otevřeným zdrojovým kódem BringAuto Logger[11], která nabízí stabilní API pro logování. Je možné s ní vypisovat informace do konzole, ukládat do souboru na vzdáleném serveru nebo do souboru lokálně.

Knihovna se stará o rotování logů v případě ukládání do souboru a také o jejich stejný formát. Rotace je velmi důležitá, protože při dlouhém sběru informací by vznikaly obrovské soubory, které by bylo těžké otevřít a zároveň by mohlo docházet ke kompletnímu zaplnění disku při velké frekvenci ukládání. Při inicializaci knihovny se pouze nastaví, jak velké mají být jednotlivé soubory s logy a kolik jich má být, zbytek už obstará sama. Interně knihovna využívá nástroj spdlog [12].

Zatím bohužel nepodporuje rotování logů při ukládání informací do více souborů, takže pokud by program Modulog měl být spuštěn po dlouhou dobu s velkým množstvím ukládaných informací, tak by bylo nutné použít argument pro zapnutí ukládání do jednoho souboru, kde knihovna rotování logů podporuje. Ukládání do adresářové struktury je v programu Modulog zatím vytvořeno pouze pro menší počty zaznamenaných informacích, ale v budoucnu, až knihovna BringAuto Logger bude podporovat ukládání do více souborů, tak tato funkcionalita bude přidána i do programu Modulog.

4.1.4 Meziprocesní komunikace

Agenti budou reprezentováni samostatně běžícími procesy, které budou vytvořeny na začátku běhu aplikace. Budou potřebovat určitým způsobem komunikovat s jádrem aplikace - při inicializaci agenta, v průběhu k posílání kontrolních zpráv a k zasílání sesbíraných informací o systému.

Ke komunikaci můžeme použít buď čtení a zápis dat do sdílené paměti, nebo kopírování dat mezi procesy. První varianta je sice rychlejší, ale celkově se nevyplatí kvůli přidané režii, která vznikne u řízení přístupu k této paměti [6]. Na následujících řádcích se nachází

[9] https://www.boost.org/doc/libs/1_56_0/doc/html/program_options.html
[10] https://github.com/jarro2783/cxxopts
[11] https://github.com/bringauto/ba-logger
[12] https://github.com/gabime/spdlog

stručný přehled druhů meziprocesní komunikace, mezi kterými byl vybírán ten nejvhodnější pro aplikaci Modulog:

Pojmenované roury

V Linuxu můžeme pojmenovanou rouru vytvořit příkazem `mkfifo nazevRoury`, čímž vznikne soubor typu pojmenovaná roura (též se jim říká FIFO). Tento soubor má stejné vlastnosti jako jiné soubory, jako např. vlastníka a povolení. Pomocí pojmenované roury poté můžeme přenášet zprávy, u kterých je zajištěna atomicita zpráv menších než velikost vyrovnávací paměti přiřazené k rouře (tato velikost může být změněna příkazem `fcntl`. Pojmenované roury neposkytují prioritu zpráv.

Fronty zpráv

Tyto fronty jsou identifikovány jménem, které musí začínat znakem „/“ a nesmí těchto znaků obsahovat více. Dají se vytvořit pomocí příkazu `mq_open()`, čímž se vytvoří záznam do virtuálního souborového systému.

Zprávy jsou označeny prioritou a nejdříve se z fronty zpracovávají zprávy s největší prioritou, následně podle času odeslání. Maximální délku zprávy je možné měnit a je uložena v souboru `/proc/sys/kernel/msgmax`.

Sockety

Sockety jsou ze všech druhů meziprocesní komunikace nejflexibilnější a jsou nejběžněji používané [6]. Představuje spojení mezi dvěma procesy a při zápisu do socketu na jednom konci jsou data přeposlána na druhý konec a obráceně.

Existují dva druhy socketů: lokální a síťové. Lokální slouží ke komunikaci na jednom počítači a síťové ke komunikaci přes síť mezi vzdálenými procesy. Ovšem síťové se dají použít i ke komunikaci na jednom počítači, pouze budou pomalejší než lokální, protože navíc budou řešit směrování a ostatní věci spojené s přenosem po síti. Za pomoci socketů se dá jednoduše implementovat i klient-server architektura, která bude ve výsledném programu potřeba. Nepodporují sice prioritní zpracování, ale to v tomto případě není důležité.

Po zvážení všech pro a proti byla nakonec vybrána komunikace přes sockety. Konkrétně knihovna `boost::asio` [13] - sice budou použity pomalejší síťové sockety, ale na druhou stranu nám knihovna ušetří spoustu času a práce oproti programování s lokálními sockety. Při použití sady protokolů TCP/IP nám vznikne i další výhoda a tou je velká přenositelnost - tato sada protokolů je standardizovaná a využívána téměř všude.

Výhodou knihovny `boost::asio` oproti zbytku sadě knihoven Boost je to, že je vyvíjena samostatně a do projektu je možno ji přidat bez přidávání zbytku knihoven Boost. Také je velmi pravděpodobné, že tato knihovna bude časem součástí C++ standardu [21].

4.1.5 Správa závislostí

V případě, kdy je projekt závislý na jakémkoliv externím zdroji, je potřeba řešit správu závislostí. Pouze stačí využít jednu libovolnou knihovnu a už je potřeba přemýšlet, jak daná knihovna bude do projektu integrována - nachází se zde totiž vždy více způsobů, jak tento problém vyřešit. Může být buď nakopírována do projektu a společně s ním se

[13] `https://github.com/chriskohlhoff/asio`

poté distribuovat, nebo se při překladu může stahovat. Existuje ale i velké množství dalších řešení.

Správa závislostí je obecně proces kompilace a správy seznamu externích entit, které jsou využívány v projektu. Tento proces může být v některých případech i velmi složitý, kdy je například potřeba řešit závislostní konflikty, což znamená, že jednotlivé části externích entit vyžadují tu stejnou entitu, ale v různé verzi.

Údržba seznamu závislostí by měla být efektivní a spolehlivá, z toho důvodu by také měla být co nejvíce automatizovaná, aby nedocházelo k manuálním chybám. Některé případy jako závislostní konflikty mohou i někdy vyžadovat manuální zásahy, ale se zbytkem akcí může velmi dobře asistovat správce závislostí.

Programovací jazyk C++ bohužel zatím nenabízí standardizovanou správu závislostí, která by byla masivně využívána téměř všemi programátory, jako je tomu např. u programovacího jazyku Java a jeho systému Maven či Gradle, nebo například Node.js a jeho systém npm. Proto je potřeba vždy najít toho správce závislostí, který se pro konkrétní případ zrovna hodí nejvíce. C++ je totiž poměrně univerzální programovací jazyk a může být využit na široké škále projektů, kde každý vyžaduje jiný přístup.

Jedním z problémů, proč je těžké spravovat v C++ závislostí je fakt, že zdrojový kód může být přenositelný mezi zařízeními, ale již zkompilovaný není. Kde oproti tomu u programovacího jazyku Java je možné kód zkompilovat do formy, která je přenositelná mezi zařízeními a poté je tedy možné tento kód používat na libovolném zařízení. U C++ se tedy nabízí řešení, kdy by byl program zkompilován pro všechny možné kombinace architektur a operačních systémů a tyto zkompilované artefakty by poté byly dodávány, ale těchto kombinací může být poměrně hodně.

Dalším problémem může být velká volnost v nastavování závislostí při kompilaci. Závislost může mít více doplňků, které se mohou před kompilací povolovat či zakazovat a poté mezi sebou jednotlivé doplňky nemusí být kompatibilní. Tento problém přidává další stupeň volnosti a přidává i na komplikovanosti správce závislostí.

Od rozumného správce závislostí by se daly očekávat hlavně následující vlastnosti:

- Jednoduché přidání nových závislostí - v případě rozšíření projektu o nové závislosti by měl být tento proces jednoduchý pro libovolnou závislost, jak již zkompilovanou, tak i nezkompilovanou
- Rychlé zjištění závislostí - uživatel by měl mít možnost rychle zjistit, jaké všechny závislosti program využívá
- Multiplatformní řešení - správce by měl být schopný dodávat knihovny jak pro různé architektury, tak i různé operační systémy
- Jednoduchá správa verzí - přecházení na nové verze knihoven by mělo být jednoduché a také by mělo být možné zařídit, aby různé projekty mohly využívat různé verze knihoven
- Možnost křížového překladu - správce by měl podporovat kompilace na architektury jiné, než ze které je projekt kompilován
- Částečná izolace od systému - projekt by měl mít možnost využívat knihovny bez jejich výchozí cesty instalace do systému.

Existující řešení

Conan

Jedná se o řešení s otevřeným zdrojovým kódem, které je založeno na decentralizovaném modelu klient-server pro ukládání závislostí - je možné je mít uloženy na vlastním serveru. Umožňuje také poskytování jak zdrojového kódu pro pozdější kompilaci, tak i již zkompilované artefakty. Lze využít napříč různými platformami a je integrovatelný do několika systémů, mezi nimiž je i CMake. Jeho nevýhodou je, že musí být nainstalován, využívá Python a není přímo integrovaný v systému CMake [3]. Také je poměrně složité naučit se vytvářet balíčky a celou logiku okolo.

vcpkg

Správce závislostí s otevřeným kódem, který je vyvíjený společností Microsoft. Je také multiplatformní a oproti správci Conan je založen na centralizovaném modelu ukládání externích balíčků. Ke každé verzi závislosti zde existuje pouze jeden balíček, což u správce Conan neplatí. Je napsán v programovacím jazyce C++ a také je integrovatelný do systému CMake [13].

Ostatní externí správci

Pro programovací jazyk C++ existuje další velké množství externích správců závislostí jako je např. Buckaroo, Hunter, Build2, CPM a jiné. Ovšem již nejsou používáni v tak velké míře, jako dva předchozí.

Kopírování závislostí

Toto řešení je velmi jednoduché, nejsou k němu potřeba žádné externí závislosti a spočívá v tom, že se celá složka se závislostí zkopíruje do projektu. Výhodou je jeho jednoduchost a možnost kompilace bez připojení k internetu, ale u větších závislostí se zbytečně zvětšuje velikost projektové složky, což je nežádoucí jev.

Git submodul

Jedním z řešení, které nevyžaduje externí závislost, je git submodul. V případě, že projekt vyžaduje externí závislost, která je dostupná v systému git, tak je možno přidat cílový repozitář do projektu skrze git submodul. Před překladem projektu je ale nutné zadat příkaz „`git submodule update --init --recursive`“, přes který se veškeré submoduly inicializují a poté se všechny tyto složky mohou přidat v systému CMake.

Výhodou tohoto řešení je jeho jednoduchost bez externích závislostí (až na git, který je ale téměř všude dostupný), ale je potřeba inicializovat nově přidané submoduly a fixace verze závislosti není příliš přímočará.

ExternalProject

Tento nástroj byl přidán do systému CMake ve verzi 3.0 a slouží k přidávání závislostí do projektu. Jeho běh se skládá z následujících kroků, kde každý z nich je samostatně konfigurovatelný:

1. Download - stažení závislosti, je možné využít jak git, tak i stažení z url adresy

2. Update - v případě znovu zavolání systému CMake je aktualizována závislost
3. Configure - konfigurace projektu, ve výchozím nastavení se předpokládá, že se jedná o projekt CMake, ale tato možnost může být přepsána
4. Build - kompilace závislosti
5. Install - instalace závislosti do předem specifikované složky
6. Test (volitelné) - spustí testy závislosti

Všechny tyto kroky jsou v systému CMake spouštěny v kompilační čase, takže v konfiguračním čase ještě nejsou k dispozici. Tento nástroj je možné využít jak pro závislosti využívající systém CMake, tak i pro ostatní.

FetchContent

Tento nástroj je v systému CMake dostupný od verze 3.11 a je podobný nástroji ExternalProject s drobnými rozdíly. Je spouštěn již v konfiguračním čase systému CMake a je uzpůsoben pro závislosti využívající systém CMake.

Knihovna CMLIB [11]

Tato knihovna je interně vyvíjena firmou BringAuto a slouží také k přidávání závislostí do projektu. Její velkou výhodou je kontrola konzistence projektu a jeho závislostí. Nevyužívá jiné závislosti než Git a CMake. Před přidáním závislostí do projektu je potřeba je mít předem zkompilované a nahrané do git LFS [14] pod názvem ve formátu `nazevKnihovny_verze_architektura-distribuce.zip`. Během přidávání závislosti v systému CMake je automaticky detekována architektura a distribuce hostitelského počítače a na základě toho je stažena předkompilovaná verze knihovny.

V programu Modulog nebyl využit žádný externí správce závislostí, protože nevyužívá velké množství závislostí a navíc většina z integrovaných je pouze hlavičkových, tudíž není potřeba je kompilovat a stačí je mít pouze nakopírované do projektové složky. Z toho důvodu byl primárně využit nástroj FetchContent, který hlavičkové knihovny pouze stáhne a ostatní po stáhnutí zkompiluje. Takto je i velmi jednoduchý křížový překlad, protože je program automaticky přeložen pro cílovou architekturu.

Je také částečně implementováno využití knihovny CMLIB, ale zatím není plně využívána, protože je stále ve fázi vývoje a určité části zatím nepodporuje (jako je například křížový překlad). Její použití je pro kompilaci mnohem rychlejší, protože není potřeba překládat závislosti a pouze se stáhnou již zkompilované.

4.2 Průběh programu

Co se týče praktické implementace, tak program na začátku svého běhu zpracuje argumenty z příkazové řádky pomocí knihovny `cxxopts` a inicializuje knihovnu BringAuto logger. Poté je vytvořena instance třídy `Core` a zavolána její metoda `start()`.

[14] Large File Storage - rozšíření systému git sloužící k ukládání větších souborů.

V ní je vytvořen TCP server pomocí knihovny `asio`, který začne čekat na nová spojení a je spuštěn v novém vlákně. Následně jsou spuštěny všechny procesy představující agenty, kteří se připojí k již vytvořenému serveru. S těmito agenty jsou také vyměněny různé konfigurační informace.

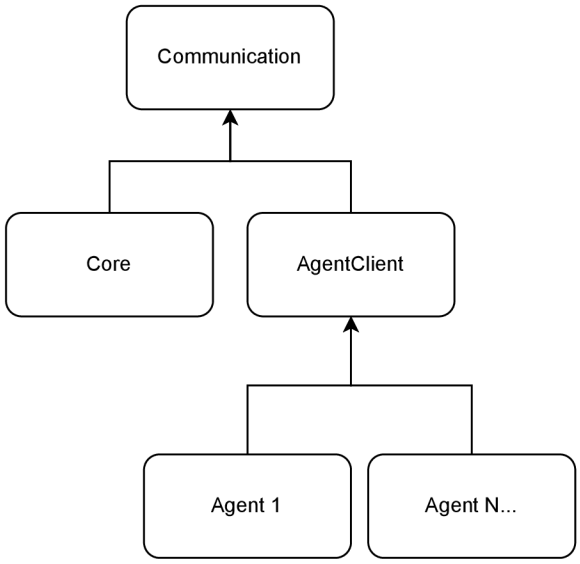
Ve vlákně, které bylo vytvořeno spolu se serverem, byly s nově navázanými spojeními vytvořeny instance tříd `TcpConnection`. V těchto instancích jsou poté asynchronně přijímány zprávy přes knihovnu `asio`. V prvních čtyřech bajtech každé zprávy je uloženo číslo, které představuje délku posílané zprávy. Ve zbytku je poté již obsah zprávy, který je v tomto případě ve formátu JSON - na efektivitu zatím nebyl brán příliš zřetel, ale do budoucna bude nutné tento způsob serializace nahradit například technologií od společnosti Google zvanou Protocol Buffers. Do každé instance třídy `TcpConnection` je předána reference na sdílenou proměnnou typu `std::condition_variable` a sdílenou proměnnou představující globální počet přijatých zpráv, které jsou poté využity v hlavním cyklu programu.

Po vytvoření všech agentů je spuštěn asynchronní časovač z knihovny `asio`, který v pravidelných intervalech agentům posílá zprávy `IS_ALIVE` a stará se o vyřešení případné neaktivity. Následně je všem agentům poslána zpráva, že mohou začít zasílat informace a třída `Core` vchází do hlavního cyklu, ve kterém zůstane, dokud nepřijde signál `SIGINT` nebo `SIGTERM`, nebo dokud nejsou všichni agenti ukončeni.

Na začátku každé iterace je vlákno uspáno, dokud není upozorněna proměnná typu `std::condition_variable` příchodem zprávy, nebo dokud existují nezpracované zprávy. Poté je iterováno skrze všechny agenty a zjišťováno, který zprávu zaslal. Poté je zpracována pouze jedna jeho zpráva z toho důvodu, aby nedocházelo při jeho velké aktivitě k tzv. „vyhladovění“ ostatních agentů.

Pokud je zpráva typu `LOG_MESSAGE`, tak je informace předána instanci třídy `LogSaver` a tam je provedeno uložení. V případě

V případě zprávy typu `CONTROL_MESSAGE` je s kontrolní zprávou naloženo přímo ve třídě `Core` náležitým způsobem podle typu zprávy.



Obrázek 4.1: Diagram závislostí

4.3 Programování vlastního agenta

Součástí aplikace Modulog je také možnost vytvářet vlastní agenty. Pro tento scénář byla vytvořena knihovna AgentClient, která vytváření velmi usnadňuje. Klienta je nutné nejdříve inicializovat pomocí funkce `initClient()`, během čehož nastane navázání komunikace s částí Core programu Modulog a vymění si konfigurační informace nutné pro běh. Poté lze využít již funkci `sendLog()` pro zasílání logů. Knihovna také nabízí funkci `sleepFor(std::chrono::seconds)`, která by měla být využívána v případě uspání klienta mezi jednotlivými sběry informací ze systému - je připravena na případ, kdy je hlavní program přerušen, tak klient toto přerušení zaznamená, ukončí čekání na sběr, informaci poskytne agentovi a ten má několik vteřin na to uvolnit veškeré prostředky a korektně se ukončit.

Struktura všech agentů by tedy měla vypadat podobně této:

```
1   auto ioContext = std::make_shared<asio::io_context>();
2   modulog::agent_client::AgentClient agentClient(ioContext, "agentId");
3   agentClient.initClient();
4   while (agentClient.canLog()) {
5       auto logMsg = std::make_shared<modulog::communication::LogMessage>(
6               modulog::communication::LogMessage::LOG_MSG_TYPE::LOG,
7               "key", "value");
8       agentClient.sendLog(logMsg);
9       agentClient.sleepFor(std::chrono::seconds(5));
10  }
11  return 0;
```

4.4 Vytvoření agenti

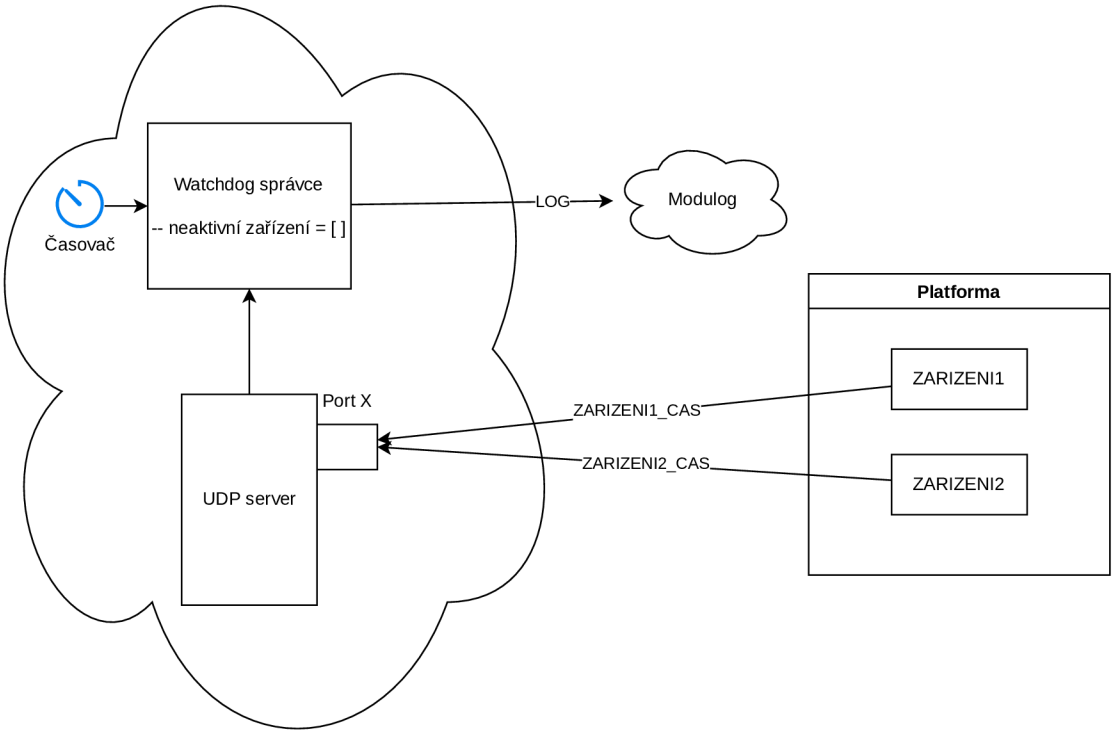
4.4.1 watchdog-agent

Autonomní vozidlo obsahuje několik prvků hardwaru, o kterých je potřeba vědět, jestli stále fungují a nenastal u nich žádný problém. Proto je potřeba, aby každý z těchto prvků pravidelně zasílal broadcastovou zprávu v předem vymezeném intervalu a někdo tyto zprávy kontroloval. A právě k tomuto lze vytvořit watchdog agenta.

Tento agent po svém spuštění začne naslouchat na předem definovaném portu, kam prvky hardwaru budou zasílat své pravidelné zprávy ve formátu `název-zařízení_časové-razítko`, konkrétně např. `GREENBUTTON1_145`. Časové razítko pro tohoto agenta nehraje žádnou roli, je pouze potřeba pro jiné části autonomního vozidla, které jsou v tomto případě irelevantní.

Agent bude mít k dispozici seznam zařízení, o kterých potřebuje vědět jejich stav. Po příchodu pravidelné zprávy si u zařízení označí, že je stále aktivní. Pokud se zařízení v časovém intervalu neozve, tak agent pošle do části core upozornění a zařízení restartuje. Pokud se zařízení i podruhé neozve, tak agent pošle chybovou zprávu.

Na obrázku číslo 4.2 lze vidět, jak použít watchdog agenta v praxi. Systém je poté přehledně rozdělený na tři části, které spolu komunikují.



Obrázek 4.2: Zakomponování watchdog agenta do celého systému

4.4.2 uptime-agent

Tento agent je velmi jednoduchý. V momentě, kdy je spuštěn zaznamená aktuální čas a poté v jednotlivých pravidelných časových intervalech zaznamenává počet vteřin od spuštění. Tento časový interval jde nastavit v konfiguračním souboru přiloženým k agentovi.

Ze zaznamenaných informací se dá poté jednoduše vyčíst, kolikrát byl agent náhodně restartován a z počtu uběhnutých vteřin také časový odhad restartu. Očekává se tedy, že restart agenta nastane pouze při restartu celého zařízení. Přesný čas restartu lze sice získat pomocí příkazu `last reboot`, ale program Modulog nabízí všechny informace přehledně na jednom místě a v lepším formátu pro případné zpracování.

4.4.3 space-agent

Tento agent slouží k monitorování volného místa na disku. V konfiguračním souboru agenta se nastaví seznam složek a diskových oddílů, které mají být sledovány. U diskových oddílů je poté pomocí funkce `getmntent` zjištěno, do jakých složek jsou oddíly připojeny a následně je v pravidelných intervalech zjišťováno, kolik volného místa ve složce zbývá. V konfiguračním souboru agenta lze také nastavit procentuální hranice zaplnitelnosti složky, která když je překročena, tak je zaznamenána chyba.

Informace o složce jsou zjišťovány pomocí standardní C++ funkce `std::filesystem::space` .

4.4.4 temperature-agent

V monitorovaném zařízení je potřeba sledovat teplotu, aby v případě poruchy bylo možné odhadnout, jestli se jednalo o selhání z důvodu přehřátí nebo důvodu jiného. Byla hledána jakákoliv C++ knihovna, která by poskytovala informaci o teplotě zařízení, ale bohužel v době psaní této práce žádná jednoduchá na integraci neexistovala. Nejjednodušší způsob zjištění teploty na zařízení s operačním systémem Linux by bylo nainstalování programu lm-sensors, což je ale zbytečná závislost navíc a poté by výstup tohoto programu spuštěného z terminálu musel být zpracován a nejedná se o příliš hezké a robustní řešení. Z toho důvodu bylo zvoleno jednodušší řešení, které předpokládá existenci souboru, ve kterém je uloženo pouze jedno číslo představující teplotu zařízení.

Téměř všechna zařízení spolupracující společnosti mají tento soubor uložený na místě `/sys/class/thermal/thermal_zone1/temp`. Tím pádem stačí pouze přečíst hodnotu z tohoto souboru a posílat ji v pravidelných intervalech na uložení do hlavní části programu Modulog. V konfiguračním souboru agenta lze také nastavit spodní a horní hranice teploty, která když je přesažena, tak je zaznamenána chyba.

4.4.5 linux-system-monitoring-lib-agent

Jedna z velmi důležitých informací, která je potřeba na zařízení monitorovat, je právě vytížení procesoru a využití RAM. Při velkém vytížení může docházet k zasekávání běžících aplikací a následně i k restartování systému. Proto je potřeba tyto informace hlídat, aby bylo možné jim zabránit a případně zpětně dohledat, zda byla chyba v zařízení způsobena právě touto cestou.

Zde se opět nabízí více možnožností, jak tyto informace získat. Bohužel neexistují funkce v C ani v C++, které by jednoduše vracely hodnoty týkající se využití procesoru, proto bude třeba zpracovávat Linuxový soubor obsahující tyto hodnoty. V zásadě se jedná o soubor `/proc/stat`, který obsahuje velké množství zajímavých informací k uložení.

V tomto souboru jsou pro potřeby využití procesoru nejdůležitější první řádky začínající slovem obsahující `cpu`. Řádek může vypadat následovně:

```
cpu0 1393280 32966 572056 13343292 6130 0 17875 0 23933 0
```

kde `cpu0` označuje konkrétní jádro procesoru a následně jednotlivé sloupce znamenají čas strávený v jednotlivých stavech. Definici všech stavů lze najít v dokumentaci, ale nejdůležitější je první sloupec, který představuje čas strávený v uživatelském módu a třetí sloupec představující čas strávený v kernel módu. Jako časová jednotka je zde ve většině případů použita hodnota $1 * 10^5$ vteřiny [8]. V souboru je také řádek začínající slovem `cpu`, který sjednocuje časy ze všech jader do jednoho.

Pokud bychom chtěli spočítat procentuálně čas strávený například v kernel módu, tak v případě, že si konkrétní čísla v řádku popisující časy nahradíme následovně:

```
cpu user nice system idle iowait irq softirq steal guest guest_nice
```

tak k získání požadované hodnoty lze provést tento výpočet:

$$(system*100)/(user+nice+system+idle+iowait+irq+softirq+steal+guest+guest\_nice)$$

Pokud by byla potřeba získat procentuálně volnou RAM, tak by se toho dalo docílit buď skrze funkci `sysinfo`, která vrací strukturu `sysinfo`, která obsahuje kromě jiných i proměnnou `freeram`, která představuje dostupnou velikost paměti [9].

Dalším velmi jednoduchým způsobem může být spuštění následujícího příkazu, který již vrátí konkrétní procentuální využití, ale je pomalejší než volání funkcí programovacího jazyka C.

```
free | grep Mem | awk '{print $3/$2 * 100.0}'
```

Výše uvedené způsoby byly nejdříve implementovány, ale následně byla objevena knihovna pro programovací jazyk C++ zvaná Linux-System-Monitoring-Library[15], která je velmi pohodlná na použití a poskytuje rovnou funkce na zobrazování procentuálních hodnot sledovaných vlastností a spoustu dalších informací, tudíž byla použita.

4.4.6 net-connectivity-agent

Tento agent slouží k zaznamenávání statusu připojení k internetu. V konfiguračním souboru agenta se nastaví interval, jak často má být prováděna kontrola připojení. Agent zaznamená stav připojení hned po jeho spuštění a poté zaznamenává pouze v momentě, kdy byla zaznamenána změna připojení. Pokud zařízení k internetu připojeno je, tak bude uložena informace s hodnotou „1“, v opačném případě hodnota „0“.

Ke zjištění stavu internetu bylo využito připojení a komunikace se stránkou `google.com`. Pokud totiž při tomto nastane jakýkoliv problém, tak to znamená problémy s připojením.

Konkrétně byla využita následující procedura pro zjištění připojení k internetu:

1. Překlad adresy `google.com` na IP adresu pomocí funkce `gethostbyname` - provede se DNS překlad, který ale na zjištění stavu připojení nestačí, protože adresa mohla být uložena ve vyrovnávací paměti. Proto je potřeba dalších kroků.
2. Převod IP adresy do binární formy.
3. Navázání spojení pomocí funkce `connect` přes port 80 - jedná se o implicitní port využívaný službou HTTP, který je na adrese otevřený.
4. Zaslání libovolného HTTP požadavku, jako je například:
   `GET /unknown/file HTTP/1.0\r\n\r\n`
5. Agent následně očekává odpověď na tento požadavek. Pokud mu z adresy dorazí jakákoliv data, znamená to, že je připojený k internetu a dokáže komunikovat s externím serverem.

Po každém kroku je zaznamenáno, zda byl proveden úspěšně či nikoliv pro rychlou pozdější analýzu problému.

4.4.7 smart-agent

Tento agent by měl sloužit ke sledování stavu disku. K tomu bohužel neexistuje knihovna napsaná v jazyce C++, proto byla potřeba použít sadu programů zvanou `smartmontools`[16]. Po instalaci je možné využívat příkaz `smartctl`, se kterým lze s různými přepínači provádět různé typy testů na zařízeních podporující technologii S.M.A.R.T. Po testu lze poté zjistit jeho výsledky a na základě nich provádět různé akce, jako zálohování disku a podobně.

[15] `https://github.com/fuxey/Linux-System-Monitoring-Library`
[16] `https://github.com/smartmontools/smartmontools`

Program nabízí také přepínač `--json`, se který je výstup programu ve formátu JSON, což je velmi jednoduché na zpracování v agentovi.

Smartmontools dokáže bezproblémově testovat disk vyskytující se v počítači, na kterém byl agent vyvíjen, ale v případě použití microSD karty ze série Utility+ od společnosti Delkin přes USB adaptér, program vypisoval následující problém:

```
Unknown USB bridge [0x05e3:0x0749 (0x1532)]
```

Z toho důvodu byla společnost Delkin zažádána o poslání vlastního softwaru pro sledování S.M.A.R.T. atributů, ale přestala odpovídat a vývoj S.M.A.R.T. agenta byl odložen na dobu neurčitou.

4.5 Instalace programu

Program Modulog bude zveřejněn s otevřeným zdrojovým kódem na webové stránce Github. Pokud by tento program chtěl někdo využít, bude mít v zásadě dvě možnosti - buď si stáhnout již zkompilovaný pro danou architekturu a operační systém, nebo si aplikaci zkompilovat sám. V jeho souboru `README.md` je podrobný popis jeho kompilace a v principu se jedná o klasickou rutinu - nejdříve se pomocí systému CMake vygenerují potřebné soubory pro překlad a následně se zavolá příkaz `make`, který se již postará o samostatný překlad.

Jelikož je tato bakalářská práce zpracovávána ve spolupráci s firmou, tak byly použity její určité programové konvence, které jdou vidět například u pojmenování CMake proměnných. Pokud bychom chtěli u programu zapnout generování cíle pro instalaci, tak je nutné tuto skutečnost specifikovat pomocí přepínače `-DBRINGAUTO_INSTALL=ON`. Po překladu je možné provést instalaci pomocí příkazu `make install` - ta spočívá v tom, že se do složky specifikované pomocí proměnné `CMAKE_INSTALL_PREFIX` nakopíruje spustitelný soubor aplikace Modulog, všechny spustitelné soubory zkompilovaných agentů včetně jejich konfiguračních souborů a nakonec také všechny použité dynamické knihovny - v tomto případě pouze knihovna na správu procesů a knihovna na ukládání logů.

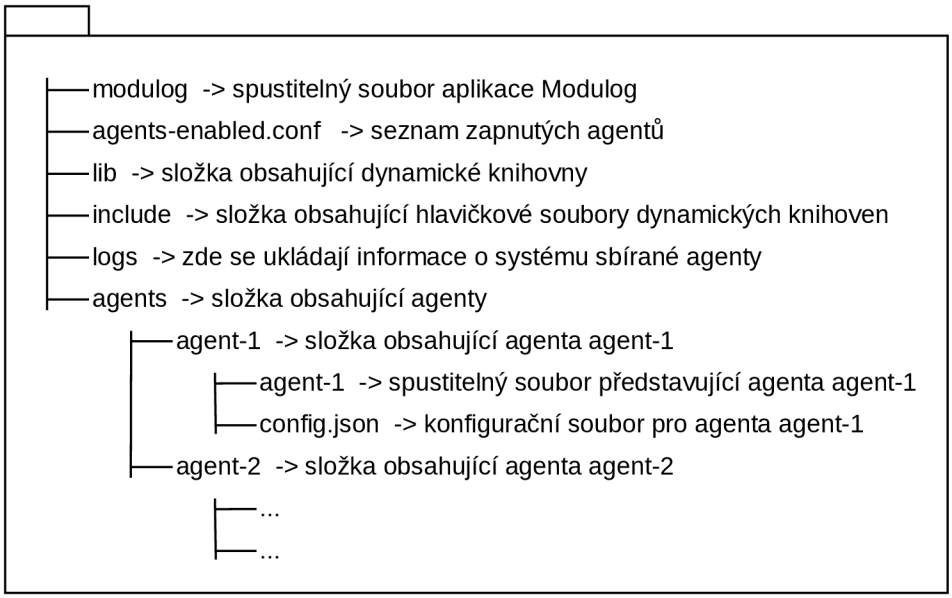
Při generování překladových souborů je také možné si v systému CMake pomocí proměnné `BRINGAUTO_SYSTEM_DEP` vybrat, jestli budou využity knihovny již nainstalované v systému, nebo se použije nástroj cmakelib [11], který stáhne knihovny předkompilované.

4.6 Vytvoření balíčku

Systém CMake nabízí možnost vytvořit zkomprimovaný balíček s výsledným programem pomocí submodulu CPack. Ten může přidat do konfiguračních souborů systému CMake novou možnost `package`, která poté pomocí příkazu `make package` vygeneruje výsledný balíček - toto lze udělat až po instalaci programu a do výsledného balíčku jsou zahrnuty všechny nainstalované soubory splňující podmínky specifikované v dokumentaci systému CMake.

Velmi důležité je při vytváření balíčku přidat do proměnné RPATH relativní cestu k dynamickým knihovnám - ve vygenerovaném balíčku programu Modulog jsou všechny dynamické knihovny uloženy ve složce `lib/`. Dynamický linker tuto informaci potřebuje vědět pro nalezení všech použitých knihoven a k následnému dynamickému linkování.

Na obrázku číslo 4.3 lze vidět strukturu výsledného balíčku.



Obrázek 4.3: Struktura balíčku aplikace Modulog

4.7 Hotové řešení

Program Modulog zatím podporuje pouze agenty napsané v programovacím jazyce C++. Bylo by možné vytvořit agenta i v jiném jazyce, ale musel by se dodržet přesný komunikační protokol a nebylo by to tak lehké jako s již existujícím klientem napsaným v jazyce C++.

Při kompilaci programu Modulog je nutné v souboru `agents-to-compile.json` specifikovat cestu ke všem agentům a poznačit u nich, jestli se mají kompilovat či nikoliv. Poté po spuštění systému CMake je tento soubor zpracován a jsou vygenerovány překladové soubory ke všem povoleným agentům i hlavnímu programu. Přes příkaz `make` je následně provedena kompilace. Soubor s povolenými agenty existuje z toho důvodu, aby bylo zabráněno zbytečně dlouhé kompilaci všech agentů v případě, kdy by byla potřeba použít například pouze jednoho agenta.

Po kompilaci všech potřebných agentů a hlavního programu je vygenerován soubor `agents-enabled.conf`, do kterého je automaticky přidán seznam všech zkompilovaných agentů. V tomto souboru lze označovat, kteří agenti budou v dalším spuštění programu Modulog použiti. Na každém řádku se nachází cesta ke složce agenta, ve které se očekává spustitelný soubor se stejným názvem, jako je název složky. Tím pádem do tohoto souboru mohou být přidáni i agenti, kteří jsou napsáni v jiným programovacím jazyce a je u nich použit správný komunikační protokol.

Po spuštění aplikace Modulog jsou spuštěni všichni povolení agenti specifikováni v souboru `agents-enabled.conf`, kteří následně sbírají informace ze systému buď v nekonečné smyčce, nebo pouze po určitou dobu na základě jejich implementace a tyto informace jsou ukládány. Aplikace běží do té doby, dokud existuje alespoň jeden aktivní agent, nebo dokud aplikaci nebyl zaslán signál `SIGINT` nebo `SIGTERM`. V druhém případě ukončí všechny agenty podle komunikačního protokolu a se ukončí sama. Všechny nasbírané informace jsou poté uloženy ve výchozí složce `logs/`, která jde z globálního nastavení přenastavit na jinou.

Jak lze vidět na obrázku číslo 4.4, tak po spuštění aplikace je pouze vypsán informační text, že agenti aktuálně zaznamenávají informace. Poté občas vypisuje určité důležité infor-

mace, jako například to, že byl některý agent nesprávně ukončen, nebo že agent neodpovídá na zprávy `IS_ALIVE`. Lze si všimnout unifikovaného formátu vypisovaných zpráv, který je vytvářen pomocí použité knihovny BringAuto Logger. Knihovna nám také vypisuje, ze kterého konkrétního procesu byla zpráva vytvořena.

```
martin@martin-HP-EliteBook-840-G7-Notebook-PC:~/School/modulog/cmake-build-debug$ ./modulog
[2022-03-31 09:47:04.417] [modulog] [info] waiting for ag. connection...
[2022-03-31 09:47:04.418] [modulog] [info] Starting new connection...
[2022-03-31 09:47:04.419] [uptime-agent] [info] Received config: NO-CONFIG
[2022-03-31 09:47:04.419] [modulog] [info] waiting for ag. connection...
[2022-03-31 09:47:04.421] [modulog] [info] Starting new connection...
[2022-03-31 09:47:04.421] [space-agent] [info] Received config: NO-CONFIG
[2022-03-31 09:47:04.422] [modulog] [info] waiting for ag. connection...
[2022-03-31 09:47:04.424] [modulog] [info] Starting new connection...
[2022-03-31 09:47:04.424] [linux-system-monitoring-lib-agent] [info] Received config: NO-CONFIG
[2022-03-31 09:47:04.425] [modulog] [info] waiting for ag. connection...
[2022-03-31 09:47:04.426] [modulog] [info] Starting new connection...
[2022-03-31 09:47:04.426] [udp-watchdog-agent] [info] Received config: NO-CONFIG
[2022-03-31 09:47:04.426] [modulog] [info] waiting for ag. connection...
[2022-03-31 09:47:04.428] [modulog] [info] Starting new connection...
[2022-03-31 09:47:04.428] [temperature-agent] [info] Received config: NO-CONFIG
[2022-03-31 09:47:04.428] [modulog] [info] waiting for ag. connection...
[2022-03-31 09:47:04.429] [modulog] [info] Starting new connection...
[2022-03-31 09:47:04.430] [net-connectivity-agent] [info] Received config: NO-CONFIG
[2022-03-31 09:47:04.430] [modulog] [info] waiting for ag. connection...
[2022-03-31 09:47:04.431] [modulog] [info] Starting new connection...
[2022-03-31 09:47:04.431] [test-random-agent] [info] Received config: NO-CONFIG
[2022-03-31 09:47:04.431] [modulog] [info] Modulog is now logging into folder ./logs/
For termination, press CTRL+C
```

Obrázek 4.4: Spuštěná aplikace Modulog

Kapitola 5

Testování

5.1 Testování v simulovaném prostředí

Aplikace bude v budoucnu nasazena do CI/CD [1] systému jako je například Teamcity [2], a proto je nutné, aby obsahovala určité typy testů. Testování aplikačních jednotek lze vynechat, protože budou vytvářeny rozsáhlejší integrační testy, které pokryjí naprostou většinu možných scénářů.

Nejdříve bylo zvažováno testování pomocí knihovny Google Test [3], ale při vynechání testů aplikačních jednotek byl vyvozen závěr, že by byla potřeba do aplikace přidat spoustu návratových hodnot pro testování všech možných typů ukončení (jako například všichni agenti byli nečekaně ukončeni, všichni agenti byli čistě ukončeni apod.).

Z tohoto důvodu bylo integrační testování pomocí knihovny Google Test zavrhnuto a přešlo se k jiné variantě: testování stavů aplikace.

Tato metoda funguje na principu, že si vytvoříme stavový diagram aplikace - jednotlivé stavy, do kterých se aplikace může dostat a přechody mezi nimi. Poté vytvoříme jednotlivé scénáře, které chceme testovat, jako například agenti posílající velké množství logů, agenti, kteří přestanou odpovídat na IS_ALIVE zprávy aj. Tyto scénáře se následně pustí a automaticky vytvoří tzv. etalony - vzorky, které popisují posloupnost stavů, kterými aplikace během scénáře prošla. Vývojář se na etalony podívá a podle posloupnosti stavů určí, jestli aplikace v daném případě běžela správně. Pokud ano, tak etalon označí za validní a příště už vůči němu může testovat aplikaci - pokud aplikace vyprodukuje stejnou posloupnost stavů, jako která je v etalonu, tak je běh považován za správný.

K testování byla využita knihovna StateSmurf [4], která splňuje všechny požadavky. Při použití je nejdříve nutno definovat již zmíněný stavový diagram aplikace, jehož návrh lze vidět na obrázku číslo 5.1.

V kódu tento návrh vypadá poté následovně:

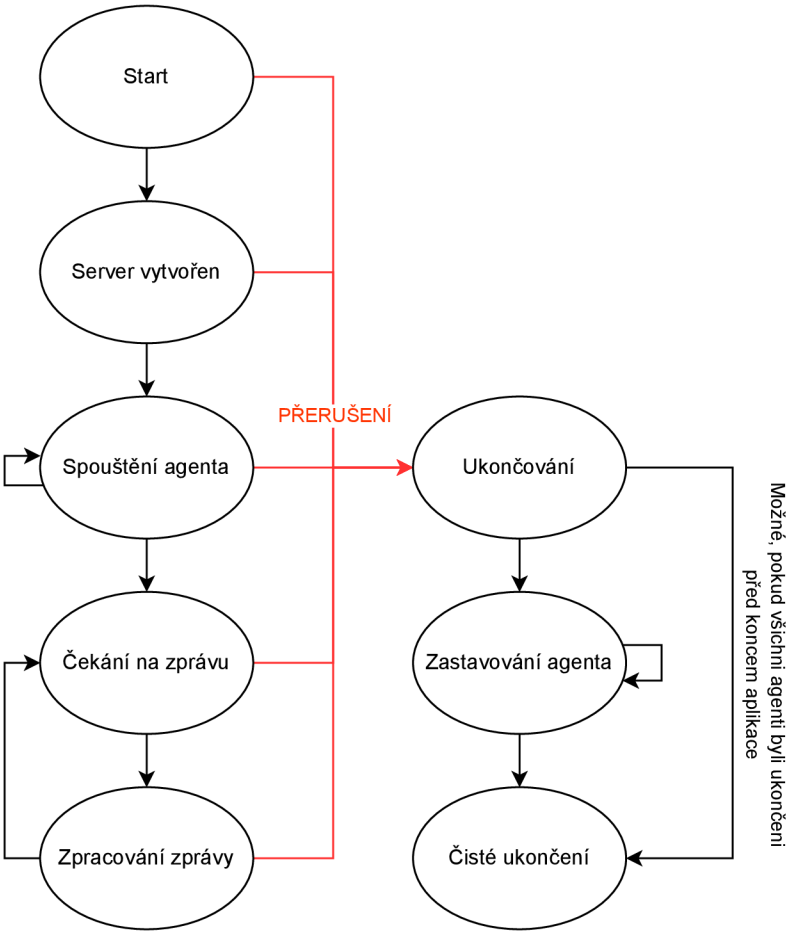
```
1 state_smurf::diagram::StateDiagram StateGraphHandler::createStateDiagram() {
2         state_smurf::diagram::StateDiagram stateDiagram;
3         auto start= stateDiagram.addVertex("Start");
4         auto serverCreated = stateDiagram.addVertex("ServerCreated");
5         auto creatingAgent = stateDiagram.addVertex("CreatingAgent");
```

[1] Systém provádějící automaticky sérii kroků nutnou pro dodání nové verze softwaru (např. automatické testování, vytvoření balíčku, nasazení).

[2] https://www.jetbrains.com/teamcity/

[3] https://github.com/google/googletest

[4] https://github.com/Melky-Phoe/StateSmurf



Obrázek 5.1: Stavový diagram

```
6         // ...
7         stateDiagram.setEdge(start, serverCreated);
8         stateDiagram.setEdge(serverCreated, creatingAgent);
9         // ...
10        return stateDiagram;
11    }
```

Dále se do globálního kontextu vloží již vytvořený stavový diagram:

```
1     auto transitions = std::make_shared
2            <state_smurf::transition::StateTransition>
3            (StateGraphHandler::createStateDiagram());
```

A poté se do aplikace přidá na konkrétní místa změna stavu následovně:

```
1     transitions->goToState("CreatingAgent");
```

Po integraci knihovny do aplikace lze vytvořit soubor `scenarios.json`, do kterého se napíší jednotlivé testovací scénáře. V našem případě to spočívá v pouštění aplikace se speciálními agenty, jako třeba agent, který se po chvíli ukončí nebo agent, který posílá obrovské množství informací. Pomocí skriptu `CompareScenarios.py` z knihovny jdou poté

generovat etalony a aplikace `SmurfEvaluator` tyto etalony může porovnávat s aktuálním výstupem aplikace.

Ke konci bylo veškeré testování aplikace vloženo do softwaru CMake, aby bylo možno aplikaci jednoduše testovat v systému CI/CD pouze pomocí příkazu `ctest` bez ohledu na to, jaký software byl pro napsání testů použit - je totiž možné, že v budoucnu se využijí i jiné metody testování než testování s použitím stavového diagramu aplikace.

Využití knihovny pro hledání chyb

Při vývoji knihovna velmi pomáhá, zvládla již upozornit na desítky chyb, které většinou byly způsobeny zapomenutím odkomentování kódu sloužícímu k zasílání zpráv typu `IS_ALIVE` a podobně. Také se několikrát stalo, že všechny integrované testy nebyly vyhodnoceny správně, a proto došlo k podezření, že se v aplikaci Modulog vyskytuje chyba typu „race condition“[5].

Při každém zhruba dvacátém spuštění testů se program ukončil s chybovým hlášením „free(): corrupted unsorted chunks“ nebo např. „malloc(): unsorted double linked list corrupted“. Všecha hlášení měla podobnou povahu, byla vypisována v podonou frekvenci a proto bylo možné odhadnout, že budou mít i stejný původ.

Největším problémem bylo, že při spouštění aplikace v debuggeru i v programu Valgrind k této chybě nedocházelo. Z toho důvodu bylo hledáno řešení, které by o chybě vypsalo více informací potřebných k nalezení příčiny. Proto byla využita kolekce nástrojů od společnosti Google zvaná Sanitizers[6]. Nabízí následující nástroje na detekce různých problémů:

- AddressSanitizer - detekuje problémy s adresováním
- LeakSanitizer - detekuje úniky paměti
- ThreadSanitizer - detekuje uváznutí a špatně synchronizovaný přístup k datům
- MemorySanitizer - detekuje využívání neinicializované paměti
- UBSan - detekuje nedefinované chování

Konkrétně byl využit nástroj AddressSanitizer, který jako jednu z možností nabízí detekci využití paměti po zavolání funkce `free()`, což bylo několikrát vypsáno programem Modulog. Pokud by tento nástroj nepomohl k objevení chyby, tak by byl využit jako druhý nástroj ThreadSanitizer, který ale nebyl nakonec potřeba.

Nástroj je součástí GCC od verze 4.8, tudíž integrace není složitá. Stačí pouze do systému CMake přidat tento řádek:

```
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fsanitize=address
                            -g -O1 -fno-omit-frame-pointer")
```

Díky němu se program přeloží s integrovaným požadovaným nástrojem AddressSanitizer. Následně stačí pouze několikrát vyvolat chybu, aby bylo dostatečné množství informací pro analýzu problému. Toho bylo docíleno využitím jednoduchého skriptu, který pouštěl v cyklu testy programu Modulog využívající knihovnu StateSmurf. Při chybném ukončení

[5] Chyba v programu, ve kterém jsou jeho správné výstupy závislé na přesném pořadí instrukcí a kde se toto pořadí může v čase měnit. Často k této chybě dochází u synchronizace procesů/vláken.

[6] `https://github.com/google/sanitizers`

programu testovací knihovna vypsala, ve kterém případě se objevil problém a nástroj AddressSanitizer se postaral o to, aby bylo vypsáno dostatečné množství informací o problému. Informace byly velmi podrobné a všechny ukazovaly na jednu proměnnou, ke které bylo přistupováno z více vláken a nebyl u ní přidán synchronizační mechanismus.

5.2 Testování v reálném prostředí

Jak již bylo zmíněno v kapitole číslo 2.5, tak ke kompilaci programů pro různé architektury nebo operační systémy se používá právě křížový překlad. Při programování aplikace v jazyce C++ je nejpohodlnější k této problematice použít systém CMake a jeho vestavěný křížový překlad. Ten má ale pro CMake určité následky [10]:

- CMake nedokáže automaticky určit cílovou platformu
- CMake nedokáže automaticky najít kompatibilní systémové knihovny a hlavičkové knihovny
- Nově vygenerované spustitelné soubory nelze spustit na hostující platformě

Z těchto důvodů je potřeba zavést tzv. „toolchain file“, což je soubor popisující způsob kompilace pro konkrétní platformu. V něm se specifikuje cílová architektura, operační systém, kompilátor a cesta ke knihovnám z cílového systému, vůči kterým potřebuje kompilátor linkovat. Dále také různé přepínače, které ale nejsou tak důležité a lze je nalézt v dokumentaci k systému CMake[7].

Pokud by byla potřeba přeložit program Modulog pro Raspberry Pi 4 model B, tak stačí pouze vytvořit následující „toolchain file“ a při kompilaci ho přiložit pomocí přepínače `CMAKE_TOOLCHAIN_FILE` k systému CMake, zbytek kompilace je již prováděn zcela automaticky.

```
1  SET(CMAKE_SYSTEM_NAME Linux)
2  SET(CMAKE_SYSTEM_PROCESSOR arm)
3  SET(CMAKE_C_COMPILER arm-linux-gnueabihf-gcc)
4  SET(CMAKE_CXX_COMPILER arm-linux-gnueabihf-g++)
5  SET(CMAKE_SYSROOT "/path/to/sysroot")
6  SET(CMAKE_FIND_ROOT_PATH ${CMAKE_SYSROOT})
7  SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
8  SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
9  SET(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
10 SET(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
```

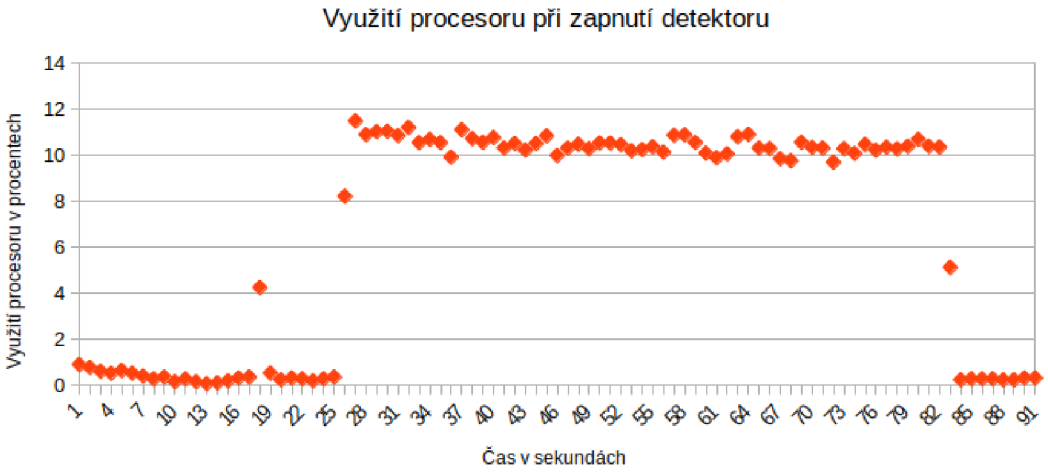
Pokud by byla potřeba, aby cílová aplikace musela být linkovaná vůči určité knihovně, tak jako jedno z jednoduchých řešeních je následující postup:

1. Knihovna se nainstaluje na cílový systém
2. Celý cílový systém se zkopíruje do složky zařízení, ze kterého je prováděn překlad
3. V systému CMake se specifikuje cesta k této složce pomocí proměnné `CMAKE_SYSROOT`

[7] `https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html`

Také se nabízí druhé řešení, kdy je knihovna předkompilována pro různé architektury, zkomprimována a přiložena k programu. Poté při kompilaci je rozbalena právě ta knihovna, která je kompatibilní s cílovým systémem a právě ta je použita.

Program Modulog byl testován na detektoru vlaků zapůjčeném od Vysokého učení technického v Brně. Tento detektor byl nasazený na zařízení Jestson Nano vsazeném do desky JN30B. Využívá kameru eCAM30 od společnosti e-con. Na tuto platformu byl program Modulog zkompilován právě křížovým překladem. Na platformě byl zapnut a bylo zjištěno, že využívá 0 - 0.6 % využití procesoru, což záleželo na tom, jestli některý z agentů zrovna zaznamenával informace, nebo byl celý program pozastavený a čekal na záznam. K tomu využíval 0.1 % paměti. Následně byly zkoumány zaznamenané informace a jestli odpovídají realitě. Na grafu číslo 5.2 můžeme vidět informace zaznamenané agentem pro měření využití procesoru, kdy ve 28. vteřině po zapnutí programu Modulog byla na detektoru zapnuta detekce a v 82. vteřině byla vypnuta. Tato skutečnost se podle reality promítnula i do měřených dat a odpovídala i výstupu programu `htop`.



Obrázek 5.2: Informace zaznamenané agentem pro měření využití procesoru

5.3 Shrnutí

Testování pomocí knihovny StateSmurf pomohlo odhalit několik chyb, které narušovaly běh aplikace při dlouhodobém běhu, a po opravení již nebyly zaznamenány žádné nové. Zavedení těchto testů také velmi pomáhá s vývojem, kdy při přidávání nových funkcionalit do aplikace lze jednoduše zjistit, jestli aplikace po zásahu nezměnila chování.

Manuální testování v reálném prostředí na detektoru vlaků bylo spíše užitečné pro ověření jednoduchosti integrace programu na samostatné zařízení. Také byla ověřována manipulace s programem, jestli jsou nasbírané informace o systému přehledné, dá se s nimi pohodlně pracovat, odpovídají realitě a podobně. Bylo také zjištěno, že program není příliš náročný na výpočetní výkon a že měřené hodnoty odpovídají realitě.

Kapitola 6

Závěr

6.1 Shrnutí

V této bakalářské práci byl vytvořen program s názvem Modulog. Jeho implementaci lze nalézt na platformě Github[1]. Slouží k zaznamenávání informací ze systému pomocí agentů běžících jako samostatné procesy. Tyto agenty lze jednoduše vytvářet pomocí C++ knihovny AgentClient, která je součástí programu. V agentech lze také nastavovat podmínky, po jejichž přerušení je zaznamenána chyba, kterou lze okamžitě vidět při nahlédnutí do zaznamenaných informací.

Program Modulog je možné využít na místech, kde je potřeba monitorovat větší množství informací, kde všechny zaznamenávané informace jsou plně konfigurovatelné. Také tam, kde je potřeba nasadit rychlé řešení bez zdlouhavé konfigurace komplexních programů, jako je např. Datadog nebo Prometheus. Dalo by se říci, že je mezistupeň mezi právě těmito komplexními monitorovacími programy a samostatnými monitorovacími programy zaměřenými na pár specifických oblastí.

Při vývoji programu byl brán zřetel na to, aby program příliš nezatěžoval systém, neměl velké závislosti a byl jednoduchý na použití. K programu jsou také přiloženy návody a prostředky pro překlad programu pro jinou architekturu, než ze které je program překládán, tudíž je možné ho jednoduše integrovat do menších zařízení bez potřebných kompilačních nástrojů.

Architektura systému byla vytvořena tak, aby byl systém téměř maximálně spolehlivý a jednotlivé části od sebe byly co nejvíce izolovány. Tak, aby selhání jednoho monitorovacího agenta nemohlo ovlivnit jiného. Zároveň jsou tato selhání zaznamenána a lze zpětně dohledat, který agent selhal. Je také kontrolováno, jestli u některého z agentů nedošlo k zaseknutí a nepřestal zaznamenávat informace.

Program byl po implementaci testován v simulovaném a reálném prostředí. K testům v simulovaném prostředí byla využita knihovna StateSmurf a toto testování odhalilo velké množství chyb. Testování v reálném prostředí bylo prováděno na detektoru vlaků a bylo zjištěno, že se s programem pohodlně pracuje, jeho naměřená data odpovídají realitě a nezatěžuje příliš systém.

[1] `https://github.com/hofin34/modulog`

6.2 Další vývoj

Jak již bylo zmíněno, tak u aplikace je kladen důraz na její modularitu a jednoduchost použití. Proto u každého jejího rozšíření je třeba klást si otázku, jestli nebude narušen její původní návrh a jestli nepřidá na celkové komplikovanosti aplikace.

Její základ by měl zůstat tudíž pořád stejný, mohou být prováděna různá drobná vylepšení v efektivitě a přidání menších funkcionalit přes argumenty příkazové řádky, ale neměly by být prováděny větší změny. Pokud by byla potřeba přidat kus kódu se zcela novým účelem, pak by bylo nutné přidat ho jako zcela samostatný balíček, který by se pouze napojil na již existující řešení v případě, že by o něho měl zrovna uživatel zájem. Jako příklad si můžeme uvést vylepšení grafické rozhraní. Aplikace Modulog v základní verzi ukládá informace pouze lokálně a to buď do složkové struktury, nebo po použití přepínače do jednoho souboru. Toto chování je dostačující, pokud je potřeba sledovat stav nevelkého počtu systémů. Při větších počtech by ale byla třeba všechna data přeposílat na jeden server, odkud by se následně dala zobrazit ve formě uživatelského rozhraní. Byla by ale třeba brát v potaz to, aby se tento doplňek mohl přidat pouze, pokud by o něj byl zájem a aby ostatní části mohly fungovat i bez něj. Pokud bychom například chtěli ukládat data jen na serveru bez uživatelského rozhraní, tak by to mělo být možné.

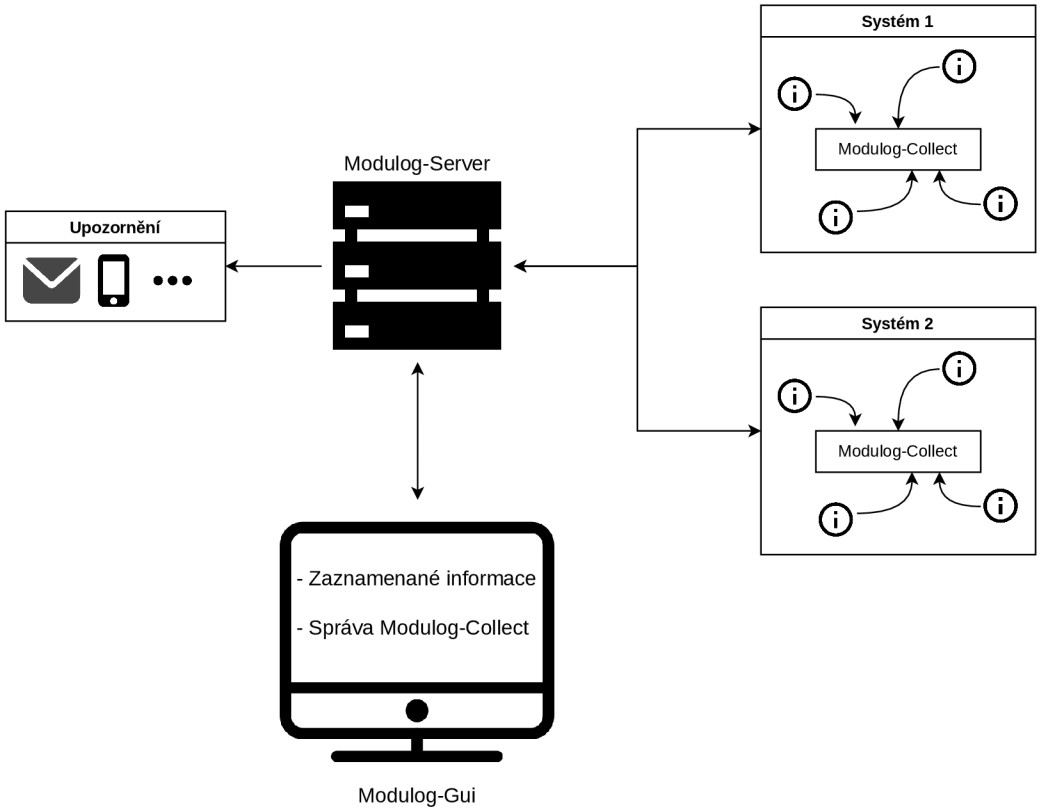
Po přidání serveru do architektury by mohlo být užitečné přidat možnost interaktivní práce s Modulogem. Aktuálně funguje na principu, že se spustí, začne sbírat data a po přerušení ukončí veškeré agenty, ukončí sám sebe a tím je ukončen jeho životní cyklus. Pokud například během sběru informací dojde k selhání libovolného agenta, tak je nutno restartovat celý program a ne pouze agenta. A to by se dalo vylepšit právě komunikací s Modulogem, kde by jeho běžící instanci bylo možno zasílat zprávy a on by na základě nich mohl provádět určité akce. Poté by bylo možné zaslat mu například zprávu typu „restartuj agenta číslo 3“ nebo zprávy podobného stylu.

Při již zmíněných vylepšeních by bylo vhodné jednotlivé části pojmenovat pro lepší čitelnost. Aktuální program na sběr informací se nazývá Modulog a při ostatních částech by se mohl označit jako Modulog-Collect. Serverová část jako Modulog-Server a grafická jako Modulog-Gui.

Při větším počtu monitorovaných systémů by bylo pohodlné také již zmíněné uživatelské rozhraní, které by oproti existujícím řešením mohlo být velmi jednoduché - na jedné stránce by měl uživatel možnost zobrazovat si stav monitorovaných systémů a na druhé by měl možnost interaktivně ovládat části Modulog-Collect na jednotlivých systémech. Pokud by byla třeba nasadit nové agenty, tak by se pouze importovali do uživatelského rozhraní a poté poslali do vybraných systémů, kde by je Modulog-Collect automaticky pustil a spravoval.

Na obrázku číslo 6.1 lze vidět předběžný návrh budoucí architektury. Bylo by také užitečné přidat do systému možnost upozornění při zaznamenání problému v jednom ze sledovaných systémů.

Také by bylo přínosné popřemýšlet nad tvorbou šablony k vytváření agentů. Aktuálně totiž tvorba spočívá v tom, že se k agentovi přiloží soubor ve formátu JSON, který se pomocí systému CMake přiloží k binárnímu souboru agenta jako konfigurační. Na začátku běhu agenta je potřeba tento JSON soubor zpracovat, uložit všechny hodnoty a následně je spuštěna rutina stejná pro všechny agenty, jako inicializace klienta a následné zaznamenávání informací. Ovšem zpracování souboru JSON je u většiny agentů velmi podobná záležitost, proto by bylo vhodné vytvořit určitou šablonu, která by všechny informace z konfiguračního souboru obsahovala. Momentálně je totiž nevýhodou to, že pokud by byla například třeba z některého důvodu změnit klíč identifikace agenta z „id“ na „name“, tak by byla potřeba



Obrázek 6.1: Předběžný návrh budoucí architektury

přepsat jak všechny konfigurační soubory, tak i veškeré agenty, protože je v nich k této informaci přistupováno individuálně přes `id = config["id"]`. S šablonou by se musely přepsat pouze konfigurační soubory a pouze jedno místo v šabloně.

Literatura

[1] Bjarne, S. *The C++ Programming Language*. 4. vyd. Pearson Education, 2013. ISBN 978-0-321-56384-2.

[2] Buyya, R., Calheiros, R. N. a Dastjerdi, A. V. *Big data: principles and paradigms*. 1. vyd. Morgan Kaufmann, 2016. ISBN 978-0128053942.

[3] conan.io. *Conan* [online]. [cit. 2022-19-04]. Dostupné z: https://github.com/conan-io/conan.

[4] Dominik Obermaier, I. S. *HiveMQ and Apache Kafka - Streaming IoT Data and MQTT Messages* [online]. [cit. 2022-26-04]. Dostupné z: https://hivemq.com/.

[5] Foundation, A. K. *Apache Kafka* [online]. [cit. 2022-26-04]. Dostupné z: https://kafka.apache.org/.

[6] Frank Vasquez, C. S. *Mastering Embedded Linux Programming*. 3. vyd. Packt Publishing, 2021. ISBN 978-1-78953-038-4.

[7] GrafanaLabs. *The analytics platform for all your metrics* [online]. [cit. 2022-21-04]. Dostupné z: https://grafana.com/grafana/.

[8] Kerrisk, M. *Proc* [online]. [cit. 2022-16-04]. Dostupné z: https://man7.org/linux/man-pages/man5/proc.5.html.

[9] Kerrisk, M. *Sysinfo* [online]. [cit. 2022-16-04]. Dostupné z: https://man7.org/linux/man-pages/man2/sysinfo.2.html.

[10] Kitware, I. *CMake* [online]. [cit. 2022-23-03]. Dostupné z: https://www.cmake.org.

[11] Kubálek, J. *CMLIB Library* [online]. [cit. 2022-16-04]. Dostupné z: https://github.com/cmakelib/cmakelib.

[12] Kumar, R. *Understanding the basics of any IoT data pipeline — from ingress capture to processing to generation of value* [online]. [cit. 2022-26-04]. Dostupné z: https://medium.com/.

[13] Microsoft. *Vcpkg* [online]. [cit. 2022-19-04]. Dostupné z: https://vcpkg.io/en/docs/README.html.

[14] Munin. *Munin* [online]. [cit. 2022-05-05]. Dostupné z: http://guide.munin-monitoring.org/en/latest/reference/munin.html.

[15] Munin. *Munin* [online]. [cit. 2022-25-03]. Dostupné z: https://munin-monitoring.org/.

[16] Murenin, C. A. Generalised Interfacing with Microprocessor System Hardware Monitors. In: IEEE. *2007 IEEE International Conference on Networking, Sensing and Control*. 2007, s. 901–906.

[17] Pinheiro, E., Weber, W.-D. a Barroso, L. A. Failure trends in a large disk drive population. 2007.

[18] Rajagopalan, R. a Varshney, P. K. Data aggregation techniques in sensor networks: A survey. 2006.

[19] Theakanath, T. K. *Datadog Cloud Monitoring*. 1. vyd. Packt Publishing, 2021. ISBN 978-1-80056-873-0.

[20] Turnbull, J. *Monitoring With Prometheus*. 1. vyd. Packt Publishing, 2018. ISBN 978-0-9888202-8-9.

[21] Wakely, J. *Working Draft, C++ Extensions for Networking*. Technical Report. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers`, 2018.

[22] Ward, B. *How Linux Works*. 3. vyd. No Starch Press, 2021. ISBN 978-1-7185-0040-2.

[23] Wojciech Kocjan, P. B. *Learning Nagios*. 3. vyd. Packt Publishing, 2016. ISBN 978-1-78588-595-2.

[24] Yip, M. *Native JSON Benchmark* [online]. [cit. 2022-16-04]. Dostupné z: `https://github.com/miloyip/nativejson-benchmark`.