FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání diplomové práce

Název:	Automatizace testů průmyslové komunikace ve virtualizovaném prostředí
Student:	Bc. Martin Štěpánek
Vedoucí:	Ing. Daniel Kubeš
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Cílem práce je úprava a rozšíření stávající testovací knihovny, která automatizuje verifikaci průmyslové komunikace, o virtualizační prvky, díky kterým bude možné za využití knihovny testovat zařízení ve virtualizovaném prostředí. Výhodou nově implementovaného rozšíření by měla být nezávislost na externích zařízeních/hardwaru a možnost simulace různých síťových topologií bez nutnosti fyzického zásahu do reálné sítě.

1. Analyzujte způsoby, kterými lze knihovnu rozšířit o virtualizační prvky a zhodnoťte je. Knihovna by měla být schopna simulovat různé síťové uspořádání a také by měla být schopna ovládat/vytvářet virtuální testovací partnery. Analyzujte také možnost propojení virtuálních testovacích partnerů se skutečnou sítí.
2. Dle výstupu z analýzy navrhněte a implementujte řešení, díky čemuž bude testovací služba schopna
a) samostatně spouštět všechny virtualizované účastníky testu,
b) vytvářet a nastavovat virtuální síť dle zvolené topologie. Testovací služba by měla podporovat sériovou topologii, topologii hvězdy a topologii kruhu.
3. Navrhněte a implementujte způsob, kterým bude možné zaznamenávat komunikaci mezi vybranými zařízeními.
4. Demonstrujte funkcionalitu vašeho řešení a zhodnoťte vlastnosti vámi implementovaného řešení. Diskutujte reálné využití daného řešení.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 17. listopadu 2022 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Automatizace testů průmyslové komunikace ve virtualizovaném prostředí

Bc. Martin Štěpánek

Katedra softwarového inženýrství
Vedoucí práce: Ing. Daniel Kubeš

4. května 2023

Poděkování

Rád bych poděkoval Ing. Danielu Kubešovi za vedení práce a veškerou práci spojenou s tím. Taktéž bych chtěl poděkovat Anně Šotolové za korekturu práce. Rovněž bych chtěl poděkovat rodině a přátelům za podporu během studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 4. května 2023 .....

Odkaz na tuto práci

Štěpánek, Martin. *Automatizace testů průmyslové komunikace ve virtualizovaném prostředí.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Abstrakt

Tato diplomová práce se zabývá analýzou, návrhem a implementací rozšíření existující testovací knihovny, která automatizuje testy verifikace průmyslové komunikace. Nová testovací knihovna implementuje integraci virtualizačního řešení Docker, s jehož pomocí je následně schopna spouštět virtualizovaná průmyslová zařízení a propojit je v různých topologiích. Součástí práce je demonstrace použití a celkové zhodnocení vytvořeného řešení.

Klíčová slova softwarové testování, automatizace testů, testovací knihovna, verifikace průmyslové komunikace, virtualizace, Docker

Abstract

This master's thesis is focused on analysis, design and implementation of extension to the existing testing library, which automates tests that verify industry fieldbus communication. New testing library implements virtualization solution Docker, which enables the library to run virtualized industry devices and connect them in numerous topologies. This thesis also includes demonstration and evaluation of the created solution.

Keywords software testing, test automation, test library, verifycation of fieldbus communication, virtualization, Docker

Obsah

Seznam obrázků

Seznam výpisů

Úvod

V dnešní době softwarového vývoje je podstatné kontinuálně, tedy co nejčastěji, testovat vyvíjený produkt. Hlavní motivací testování je snaha neustále zvyšovat kvalitu kódu a tedy daného produktu. Čím dříve je totiž nalezen daný problém v softwaru, tím rychleji ho lze lokalizovat a opravit. Toto dřívější objevení poté může vést i k menšímu objemu kódu, ve kterém je potřeba zdroj chyby hledat. Zároveň neobjevení chyby v průběhu vývoje může mít negativní dopady na firmu, která daný produkt vytváří - od poklesu reputace až po finanční ztrátu.

Tento negativní dopad může být ještě markantnější v kontextu průmyslových zařízení. Zákazník od těchto zařízení očekává skoro nepřetržitý běh a vysokou stabilitu, protože každá minuta, kdy zařízení neběží tak, jak má, přináší jeho majiteli finanční ztrátu. V horším případě může chyba způsobená špatně fungujícím zařízením zapříčinit poškození vybavení, nebo dokonce zranění obsluhy výroby.

Testování ovšem bývá často repetitivní činnost, proto se k testování v dnešní době hojně využívá automatizace. Díky ní jsme schopni častěji testovat, minimalizuje se faktor lidské chyby a jsme schopni lépe využít lidské zdroje, které by jinak musely manuálně testovat. Firmy proto často investují nemalé peníze do automatizace testů a jako každý ziskový subjekt se snaží svoje zisky zvyšovat snižováním vlastních nákladů.

Jak tedy může firma snížit svoje náklady na testování? Jednou z možností je snížení počtu zúčastněných fyzických zařízení. Každé fyzické zařízení musí být zakoupeno a musí být spravováno. Zařízení jsou následně sestavována v různých topologiích. Je tedy potřeba mít dostatek zařízení, abychom pokryli všechny potřebné konfigurace. Tyto sestavy zařízení následně musí být někým spravovány a každá změna v topologii může zabrat nemálo času a úsilí.

Možností, jak tento problém vyřešit, je virtualizace testovacích zařízení a topologie zapojení těchto zařízení. Díky virtualizaci jsme schopni simulovat jakékoliv zařízení v různých počtech. Cenou přidání dalšího zařízení jsou ná-

sledně pouze nutné výpočetní zdroje na zařízení, na kterém testování běží. Dostatečné výkonné zařízení může samozřejmě stát nemalé finance, ale takovéto zařízení je univerzálnější a znovu použitelné pro všechny produkty, které podporují virtualizaci.

Tato práce se věnuje návrhu a implementaci rozšíření stávající testovací knihovny, která bude podporovat virtualizaci zařízení. Knihovna by zároveň měla podporovat propojení zařízení v několika různých topologiích. Knihovna je vytvářena pro společnost Siemens s.r.o., která ji následně využije při vývoji svých produktů.

Práce začíná kapitolou 1, ve které stanovuje cíle práce. V kapitole 2 je přiblížen teoretický kontext této práce. Následně v kapitole 3 práce analyzuje stav stávající testovací knihovny a analyzuje možnosti rozšíření knihovny. Kapitola 4 se poté věnuje návrhu změn a rozšíření v testovací knihovně, jejichž implementace je následně popsána v kapitole 5. V kapitole 6 je následně ukázáno použití testovací knihovny. V neposlední řadě se kapitola 7 věnuje zhodnocení nové testovací knihovny.

Kapitola 1

Cíl práce

Cílem této práce je úprava a rozšíření stávající testovací knihovny, která automatizuje testy verifikace průmyslové komunikace, o virtualizační prvky, díky kterým bude možné za využití knihovny testovat zařízení ve virtualizovaném prostředí. Součástí by měla být i analýza, podle které bude zvolena nejvhodnější možnost implementace testovací knihovny.

Výhodou nově implementovaného rozšíření by měla být nezávislost na externích zařízeních/hardwaru a možnost simulace různých síťových topologií bez nutnosti fyzického zásahu do reálné sítě.

Jednotlivé úkoly tedy jsou:

1. Analyzovat způsoby, kterými lze knihovnu rozšířit o virtualizační prvky a zhodnotit je. Knihovna by měla být schopna simulovat různé síťové uspořádání a také by měla být schopna ovládat/vytvářet virtuální testovací partnery. Součástí by měla být také analýza možnosti propojení virtuálních testovacích partnerů se skutečnou sítí.

2. Dle výstupu z analýzy navrhnout a implementovat řešení, díky čemuž bude testovací služba schopna

   a) samostatně spouštět všechny virtualizované účastníky testu,

   b) vytvářet a nastavovat virtuální síť dle zvolené topologie. Testovací služba by měla podporovat sériovou topologii, topologii hvězdy a topologii kruhu.

3. Navrhnout a implementovat způsob, kterým bude možné zaznamenávat komunikaci mezi vybranými zařízeními.

4. Demonstrovat funkcionalitu řešení a zhodnotit vlastnosti implementovaného řešení. Součástí by měla být i diskuze nad reálným využitím daného řešení.

Kapitola 2

Teoretická část

Tato kapitola je věnována přiblížení teoretického kontextu této práce.

2.1 Virtualizace

Koncept virtualizace se poprvé objevil na konci 50. let minulého století, kdy skupina z University v Manchesteru vytvořila první funkční prototyp virtuální paměti. Následně v 60. letech minulého století byl koncept rozšířen i na celé počítače. První tzv. *virtual machine* (VM), neboli česky „virtuální stroj", byl vytvořen firmou IBM a jeho cílem bylo umožnit souběžný přístup k sálovým počítačům. Každá VM byla instancí fyzického stroje a dávala iluzi toho, že uživatel přistupuje přímo k fyzickému stroji. Uživatelé mohli vyvíjet, spouštět a testovat aplikace bez obav z toho, že by se mohl zhroutit celý počítač, díky tomu, že každá VM byla izolovanou kopií systému. [1]

V polovině 70. let minulého století byla virtualizace již dobře akceptovaným konceptem mezi uživateli. Použitím virtualizace totiž mohli vyřešit podstatné problémy této doby. Například díky již zmíněné virtualizované paměti mohli uživatelé adresovat mnohem větší operační paměť, než počítač skutečně obsahoval. [1]

Všechny tyto virtualizační techniky byly primárně cestou k vyrovnání se s vysokou pořizovací cenou hardwaru v této době. Díky tomu mohli majitelé sálových počítačů využít svoji investici co nejefektivněji. Jejich použití se tedy se snižujícími cenami hardwaru začalo snižovat a dokonce skoro vymizelo během 80. a 90. let díky příchodu levnějších osobních počítačů. [1][2]

Poptávka po virtualizaci začala znovu růst až v 90. letech minulého století. Se zvětšující se různorodostí hardwaru a softwaru zde vznikla poptávka na schopnost spustit aplikace, které byly původně určeny pro jiný hardware a jiný operační systém, na jednom určitém stroji. [2]

Poptávka se ovšem zvětšila i v komerčním sektoru. S zvyšujícími se počty serverů firmy hledaly způsoby, jak využít tuto investici do serveru naplno.

Spustit pouze jednu aplikaci na serveru nebylo příliš ekonomické, jelikož tato aplikace nemusela vždy využít všechny výpočetní zdroje, a nákup dalších serverů přinášel mimo samotné pořizovací ceny i další náklady, například na údržbu a administraci. Řešením tohoto problému byla opět virtualizace. Oba tyto trendy pokračují dodnes a patří mezi hlavní důvody pro dnešní využití virtualizace. [1][2]

Virtualita se od reality liší pouze ve formálním světě. Má ovšem podobné jádro nebo efekt. V počítačovém světě je virtualizované prostředí vnímáno aplikací stejně jako reálné prostředí, přestože některé mechanismy v něm mohou fungovat jinak. Toto prostředí hlavně dává aplikaci zkreslený obraz reálného stroje, přičemž v tomto zkresleném obraze může mít stroj více či méně určitých zdrojů. Typický moderní počítač využívá spoustu takovýchto přístupů. Jedním příkladem je již zmíněná virtuální paměť, díky níž může proces použít mnohem více paměti, než je fyzicky dostupné. Tato virtuální paměť taktéž umožňuje sdílení fyzické paměti mezi stovkami procesů. Podobným konceptem je i multitasking, kdy jeden procesor je rozdělen a prezentován jako „virtuální procesor“ jednotlivým procesům. Na druhou stranu, několik procesorů může být seskupeno do jednoho výkonnějšího virtuálního procesoru. [2]

Možností virtualizace je tedy spousta, a to na mnoha úrovních. Virtualizaci tedy můžeme nadefinovat takto:

> Virtualizace je technologie, která kombinuje/rozděluje, výpočetní zdroje k prezentaci jednoho či více prostředí za pomoci metodologií jako hardwarového a softwarového rozdělování/agregování, parciální/kompletní simulací stroje, emulace a spousty dalších. [2]

Jak je z definice vidět, virtualizace není jen o rozdělování výpočetních zdrojů, ale i o jejich sjednocování do větších celků. V rámci této práce se ale budeme zaměřovat primárně na rozdělování výpočetních zdrojů za pomoci virtualizace. Mezi praktické využití virtualizace může patřit například:

Konsolidace serveru Za pomoci virtualizace lze přerozdělit práci z více serverů, které nejsou na plno používány, na jeden server, a tím ušetřit na hardwaru, managementu a administraci infrastruktury. Díky ní tedy snižujeme komplexitu administrace, jelikož veškerý software ve VM je nezávislý na softwaru fyzického serveru. [2]

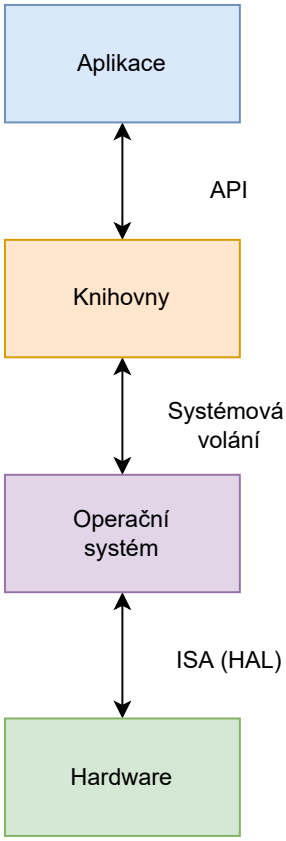
Sandbox Za pomoci virtualizace lze vytvořit bezpečné a hlavně izolované prostředí pro běh aplikací, jejichž původ nám může být neznámý a spuštění v normálním prostředí počítače by mohlo představovat bezpečnostní riziko. Tato technika se označuje jako *sandboxing*. [2]

Virtuální hardware Virtualizace může zprostředkovat hardware, který počítač nikdy neměl. Mezi ně patří například virtuální ethernetové adaptéry, switche, huby atd. [2]

Spuštění více OS Bez virtualizace nejsme schopni spustit více operačních systémů zároveň. Virtualizace tedy přináší prostředky, jak toto umožnit. Mimo to ale také přináší prostředky, jak spustit staré operační systémy na novém hardwaru, pro který daný operační systém nemusel být vyvinut. [2]

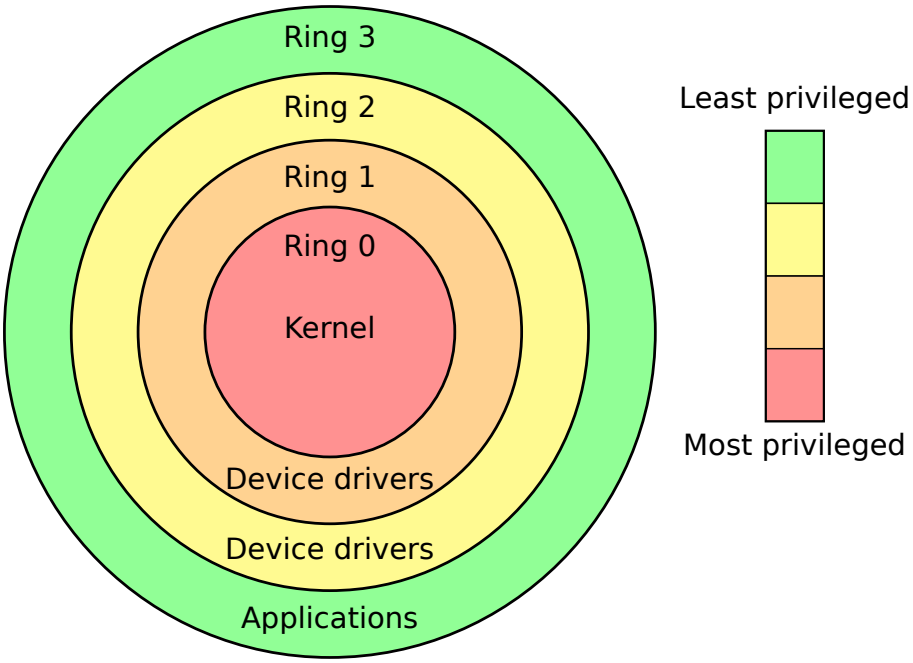
Testování/QA S pomocí virtualizace jsme schopni vytvořit libovolné testovací scénáře, které jinak mohou být těžce simulovatelné v reálném světě. [2]

Jak je vidět z tohoto krátkého seznamu, důvodů proč virtualizovat je spousta. Jak ale virtualizace funguje? V praxi je virtualizace abstrakční vrstva, která poskytuje potřebné propojení mezi dvěma izolovanými vrstvami. Toto si lze lépe představit pomocí obrázku 2.1, na kterém lze spatřit zjednodušenou architekturu počítače. Je zde vidět, že virtualizační vrstvu lze vložit mezi jakékoliv dvě vrstvy a tím docílit virtualizace. [2]



Obrázek 2.1: Architektura počítače ukazující příležitosti k virtualizaci. Podle Nanda Susanta et al. [2]

V případě dnes nejpopulárnější architektury x86 je potřeba i rozlišit, s jakými právy daná virtualizace běží. Architektura x86 definuje tzv. *protekční kruhy*, kde každý kruh(anglicky se tyto úrovně označují jako *ring*), neboli úroveň, definuje, co daná aplikace může a nemůže udělat. Grafickou představu těchto úrovní můžeme vidět na obrázku 2.2. Jak je vidět, čím nižší úroveň, tím více práv software běžící v této úrovni má. Typicky v úrovni 0 běží operační systém, v úrovních 1 a 2 systémové ovladače, a v úrovni 3 tři samotné aplikace. [3]



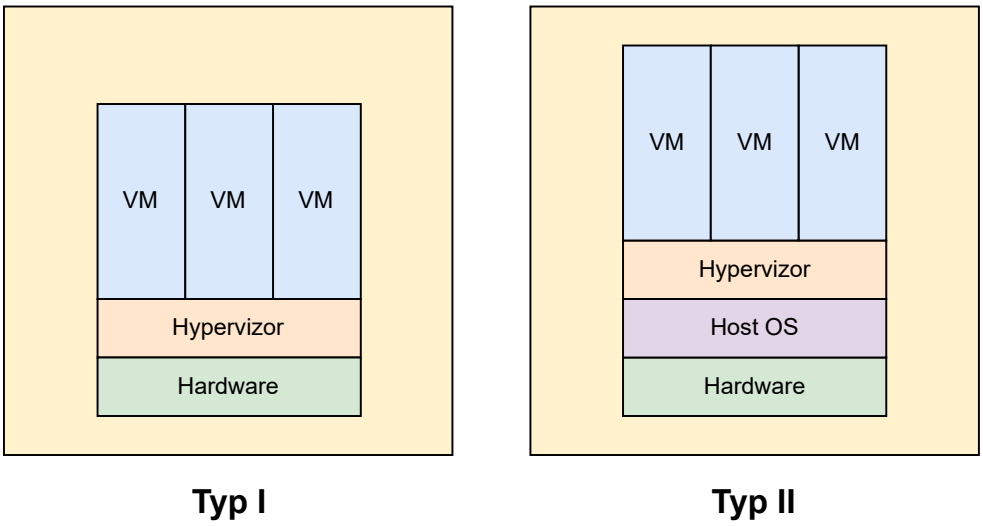
Obrázek 2.2: Protekční kruhy na architektuře x86. Zdroj Wikipedia Commons [4]

2.2 Hypervizor

Důležitou komponentou při virtualizaci je tzv. *hypervizor* (někdy také označován jako Virtual Machine Monitor - VMM). Hypervizor je softwarová vrstva, která virtualizuje všechny potřebné zdroje fyzického stroje, a tím definuje a podporuje běh jednoho či více VM. [5]

Existují dva typy hypervizorů. Ty můžeme vidět na obrázku 2.3. Hypervizor typu I běží přímo nad hardwarem a spravuje všechny virtuální stroje. Díky tomu hypervizor může mít plnou kontrolu nad hardwarem, jelikož běží v úrovni nejvyšší priority. Hypervizor typu 2 běží nad hostitelským operačním systém, respektive uvnitř něj. Tento hypervizor je odkázán na hostitelský

systém, který mu přiděluje prostředky, jelikož každá VM je v podstatě další proces v systému. [2][3]



Obrázek 2.3: Typy hypervizorů. Podle Fernando Rodríguez-Haroa et al. [3]

2.3 Možnosti virtualizace z technologického pohledu

Jak již bylo nastíněno, virtualizační vrstva může existovat na více úrovních. V následujících sekcích bude přiblížena virtualizace od nejnižší vrstvy až po tu nejvyšší.

2.3.1 Virtualizace ISA

ISA, neboli *Instruction Set Architecture* je součástí abstraktního modelu počítače a definuje, jak může software ovládat procesor. ISA funguje jako rozhraní mezi hardwarem a softwarem a specifikuje, co a jak je procesor schopen udělat. [6]

Virtualizace na této úrovni tedy funguje za pomoci softwarové emulace ISA dané platformy. Tedy virtuální vrstva musí být schopna přeložit ISA instrukce hosta na ISA instrukce hostitele. Tento způsob lze označit i jako *binární překlad*. Tato virtualizace funguje pouze v případě, že existuje způsob, jak na hostitelské platformě provést všechny úkony požadované architekturou hosta. [2]

Výhodou této virtualizace je, že nám teoreticky dokáže dát dostatečné prostředky ke spuštění jedné platformy (jako například x86) na ostatních platformách. Taktéž jsme schopni spustit operační systémy bez jakékoli modifikace.

Tato portabilita ovšem přichází s nemalou cenou. Tím, že musíme každou instrukci softwarově přeložit, následně dochází k velkému snížení výkonu systému. I tak si tento způsob virtualizace však nalezne využití. [2][3]

2.3.2 Virtualizace HAL

Hardware abstraction layer (HAL), neboli česky hardwarová abstrakční vrstva, je vrstva, která vytváří jednotné API pro přístup k hardwaru. Díky ní je možné oddělit implementaci přístupu k hardwaru od samotného softwaru. Software totiž pouze využívá API definované HAL a až následná implementace na daném zařízení definuje reálný přístup k hardwaru daného stroje. [7]

Virtualizace HAL tedy využívá podobnosti mezi architekturou platformy hosta a hostitele, díky čemuž je snížena latence způsobená překladem. Často je ovšem spojována společně s binárním překladem. Ten může nastat například z důvodu nedostatečných práv k vykonání operace v rámci úrovně, ve které hypervizor běží. [2][8]

2.3.3 Virtualizace na úrovni programovacího jazyka

Virtualizace na úrovni programovacího jazyka probíhá v rámci aplikační vrstvy. Ideou bylo vytvořit virtuální stroj již na aplikační úrovni, který oddělí aplikaci od hardwaru. Implementace tohoto aplikačního virtuálního stroje následně definuje reálný přístup k výpočetním zdrojů. K těm je přistupováno standardizovaným způsobem, jenž je definován abstrakční vrstvou. Pro představu lze uvést například Java Virtual Machine, nebo Common Language Runtime v .NET Frameworku. [2]

2.3.4 Virtualizace na úrovni knihoven

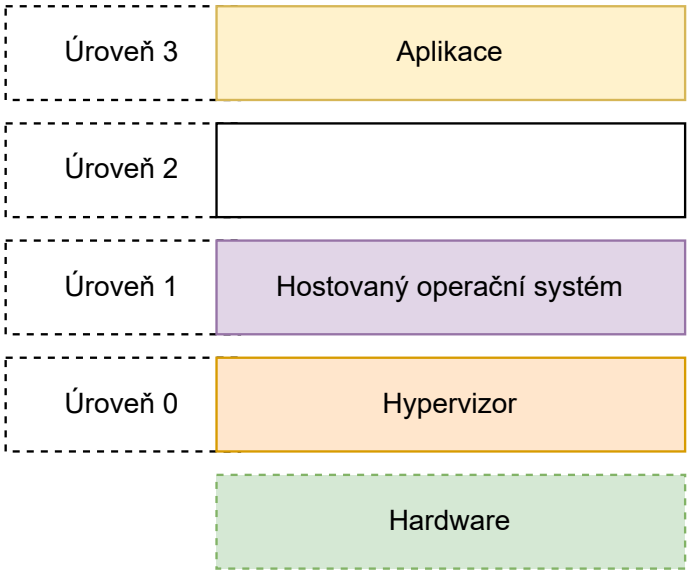
Virtualizace na úrovni knihoven může připomínat předchozí popsanou virtualizaci, ovšem důležitá je změna původu této implementace. Každá aplikace využívá volání různých externích zdrojů. Tyto zdroje mohou a nemusí být přenositelné mezi operačními systémy. Virtualizace na této úrovni využívá rozhraní daných externích zdrojů, ovšem implementace je přizpůsobena tomu stroji, na kterém aplikace běží. Jako příklad lze uvést například Wine, který implementuje Windows API v UNIX prostředí. [2]

2.4 Virtualizační techniky

V následující sekci jsou přiblíženy reálné přístupy k virtualizaci, které používají buď jednu z předem zmíněných virtualizačních technik, nebo tyto techniky kombinují.

2.4.1 Plná virtualizace

Plná virtualizace funguje na principu kombinace přímého spuštění instrukcí a binárního překladu instrukcí za běhu. Využívá se toho, že hypervizor běží na nulté úrovni, tedy s nejvyšší prioritou a s největšími právy. Hypervizor tím pádem zcela odděluje operační systém od skutečného hardwaru. Toto lze vidět na obrázku 2.4. [9]



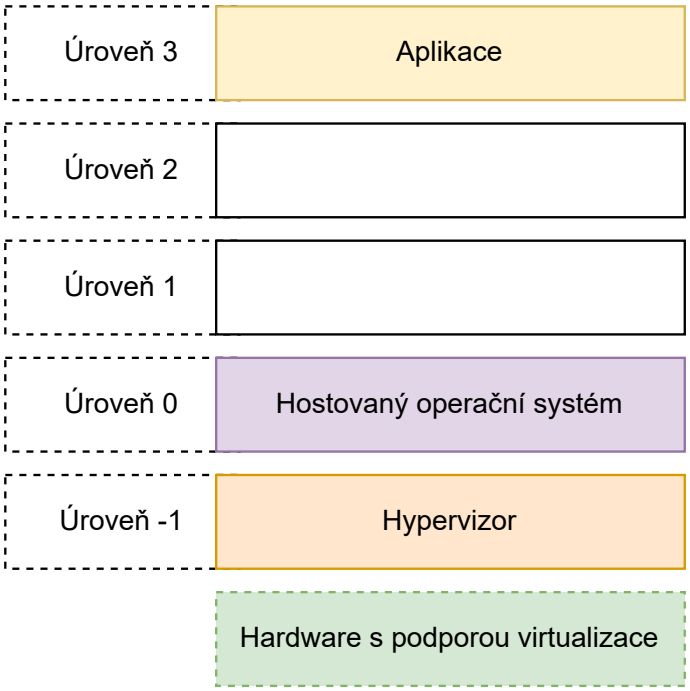
Obrázek 2.4: Plná virtualizace. Podle Jana Fesla [8]

2.4.2 Paravirtualizace

Paravirtualizace funguje na stejných úrovních jako plná virtualizace. Hypervizor má plnou kontrolu nad hardwarem hostitelského stroje a operace s ním jsou zprostředkovávány hypervizorem. Změna ovšem nastává v rámci operačního systému. Ten má totiž upravené jádro a díky tomu ví, že běží v paravirtualizovaném prostředí. Jednotlivá ISA volání jsou tedy nahrazena voláním virtualizačních funkcích. Tedy operační systém přímo komunikuje s hypervizorem. [9]

2.4.3 Hardwarově asistovaná virtualizace

Hardwarově asistovaná virtualizace funguje na principu přímé podpory virtualizace ze strany hardwaru. V rámci této virtualizace je mezi protekční vrstvy přidána nová vrstva na úroveň -1, tedy pod všechny ostatní úrovně. Privilegované instrukce jsou tedy zachyceny přímo hardwarem a následně jsou předány hypervizoru ke zpracování.[9]



Obrázek 2.5: Hardwarově asistovaná virtualizace. Podle Jana Fesla [8]

2.4.4 Kontejnerová virtualizace

Kontejnerová virtualizace, neboli zkráceně kontejnerizace, je lehčí alternativou k tradiční virtualizaci. Porovnání s tradiční virtualizací můžeme vidět na obrázku 2.6. Tradiční VM jsou nahrazeny tzv. *kontejnery*. Jak je vidět, kontejnery již nemusí obsahovat kopii celého operačního systému. Kontejnery totiž sdílí jádro operačního systému a další zdroje, jako například knihovny. Díky tomu mohou být spouštěny mnohem rychleji než VM. [10]

Všechny kontejnery jsou následně orchestrovány tzv. *kontejnerizátorem*, v angličtině *container engine*. Ten má na starost vše spojené s procesem spouštění a běhu kontejneru. Díky němu je mimo jiné možné rychle a efektivně propojit kontejnery mezi sebou. [11]

To, co kontejnery činí tak efektními, je však zároveň i jejich nevýhodou. Sdílením jádra může docházet k bezpečnostním rizikům, kvůli čemuž jsou méně bezpečné než tradiční VM. [12]

Nástroje, které umožňují kontejnerovou virtualizaci můžeme dle Aleksic [13] rozdělit do čtyř kategorií:

Image builders Nástroje, které umožňují vytvářet kontejnery v souladu s pravidly dle OCI.

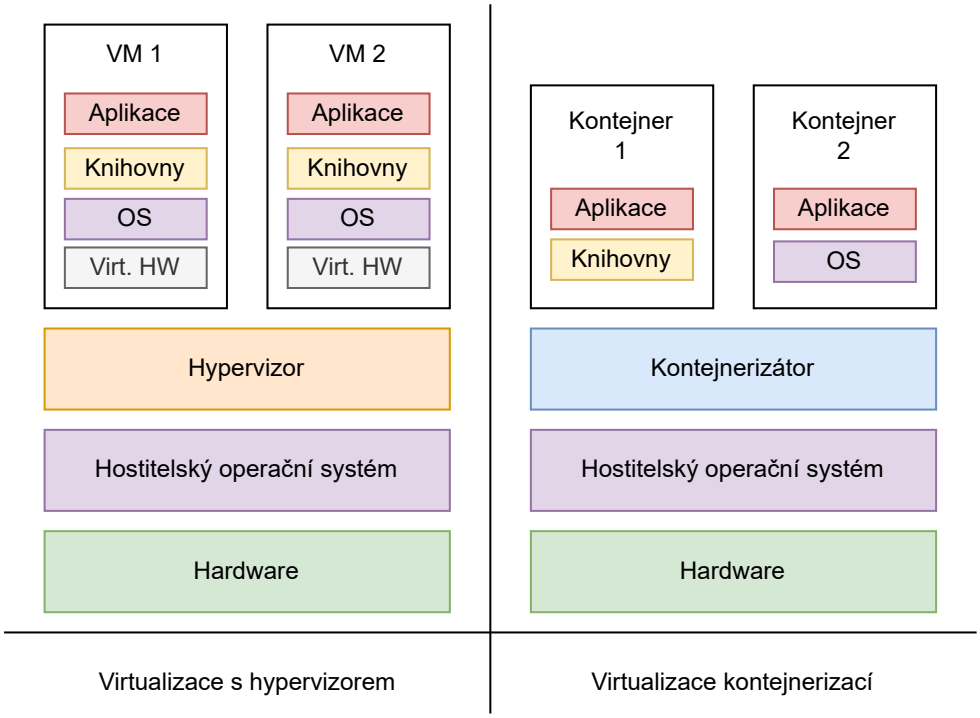
Container managers V širším smyslu tak můžeme nazvat všechny nástroje, které pomáhají se sestavováním a během kontejnerů. Specificky ale tak

můžeme nazvat ty nástroje, které pomáhají spravovat jednotlivé instance kontejnerů.

Container runtimes Nástroje, které se starají o běh kontejnerů.

Container engines Nástroje, které se starají o všechny procesy spojené s kontejnerizací.

K úspěšnému využití kontejnerizace je samozřejmě potřeba spravovat veškeré procesy spojené s kontejnerizací.



Obrázek 2.6: Porovnání virtualizace s hypervizorem oproti virtualizaci kontejnerizací. Podle Jana Fesla [8]

2.5 Virtualizační řešení

V předchozích sekcích byly představeny možnosti, jak lze virtualizovat. Následující sekce jsou věnované vybraným řešením, které dnes pro virtualizaci existují.

2.5.1 QEMU

QEMU je emulátor, který využívá dynamického binárního překladu. Tento nástroj podporuje dva operační módy:

1. Emulace uživatelského prostoru
2. Emulace celého systému

Za pomoci prvního módu je QEMU schopno spustit aplikaci, která byla zkompilována pro jeden procesor na jiném procesoru. V druhém emuluje celý hostovaný systém. QEMU podporuje velikou řadu procesorových architektur, jako například x86 nebo ARM. [2]

QEMU funguje tak, že za běhu překládá instrukce do tvaru, který vyžaduje hostující systém. Hlavní ideou je, že každá instrukce je rozdělená na několik jednodušších instrukcí, které se nazývají *micro operations*, neboli česky mikro operace. Každá tato zjednodušená instrukce je implementována kusem kódu napsaném v jazyce C a následně zkompilována. [2][14]

Výstupní objektový soubor je následně použit jako vstup pro vytvoření dynamického generátoru kódu. Ten je následně spuštěn a spojením několika mikro operací dokáže vygenerovat plnohodnotnou instrukci pro hostitelský stroj. [2][14]

2.5.2 VMware

VMware je dnes známá technologická firma se širokou škálou produktů. To, co ale zpopularizovalo tuto firmu a její produkty, je virtualizace. Firma VMware podporuje oba dva typy hypervizorů, jež jsou k vidění na obrázku 2.3.

Implementace hypervizoru typu I se nazývá VMware ESXi a zaměřuje se na komerční virtualizaci serveru a jím poskytnutých služeb. Dle definice typu hypervizor běží přímo nad hardwarem a tedy i přímo s ním komunikuje. Všechny VM následně běží nad tímto hypervizorem. [15]

Hypervizor typu II. poté můžeme nalézt v produktu VMware Workstation. Tento produkt je zaměřen na virtualizaci na osobním počítači. Produkt, který je nainstalován v prostoru hostujícího operačního systému, je spouštěn jako kterýkoliv jiný program. Jeho součástí ovšem je předem zmíněný hypervizor, který se stará o propojení mezi VM a hardwarem. Komunikaci s hardwarem potom hypervizor realizuje za pomoci systémových volání hostujícího operačního systému. [16]

Oba dva tyto produkty pracují na pomocí virtualizace celého operačního systému, tedy všechny VM obsahují plnohodnotný hostovaný operační systém.

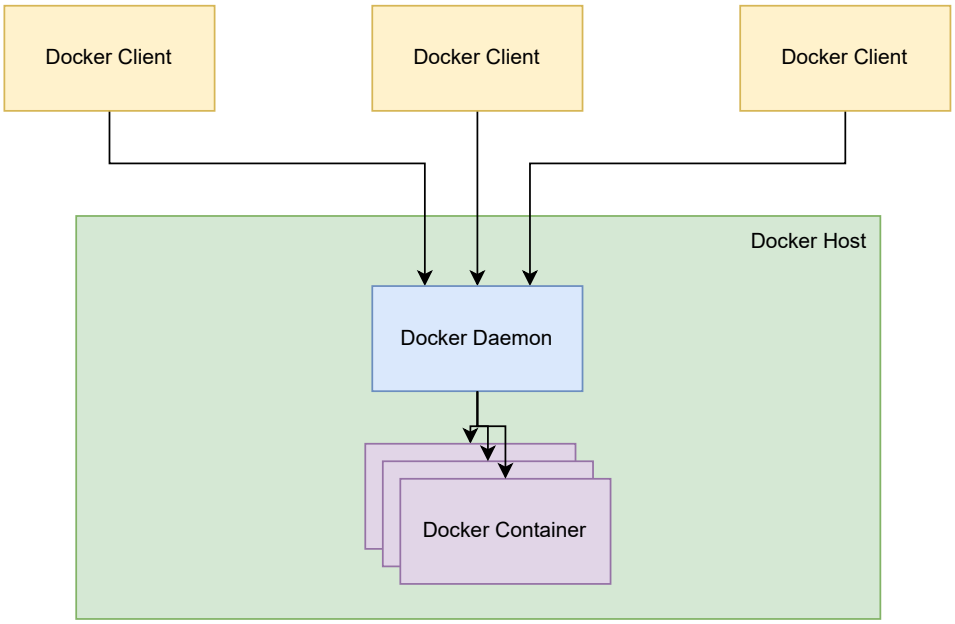
2.5.3 Docker

Docker je neznámějším a nejrozšířenějším nástrojem, který zprostředkovává kontejnerizaci. Proto je také ve většině případů většina ostatních nástrojů s Docker srovnávána. Docker se skládá z několika hlavních komponentů, které dohromady tvoří container engine. Mezi tyto komponenty patří

- Docker Client and Server,

- Docker Containers,
- Docker Images,
- Docker Registries.

Docker Client and Server, jak z překladu názvu vyplývá, obsahuje klienta a server. Server dostává žádosti od klienta, které následně zpracovává. Architekturu nástroje Docker můžeme vidět na obrázku 2.7. Důležitou komponentou je Docker Daemon, což je persistentní proces, který neustále běží na pozadí. Klient provádí komunikaci s Docker Daemon prostřednictvím definovaného REST API a ten se následovně stará o sestavování, běh a distribuci Docker kontejnerů. Docker server a klient mohou běžet na jednom stroji, ale zároveň je možné se za pomocí klienta připojit ke vzdálenému serveru. [17][18]



Obrázek 2.7: Docker architektura. Podle James Turnbull [17]

Jednotlivé Docker kontejnery jsou vytvářeny z tzv. Docker Images. Tyto images, neboli v češtině obrazy, obsahují definici všech závislostí a balíčků, potřebných k běhu kontejneru. Způsoby jak vytvářet obrazy jsou dva - buď můžeme definovat vlastní obraz, anebo použít nějakou již vytvořenou předlohu. Nutno podotknout, že každý existující obraz může být předlohou pro nový obraz. [17]

Tyto obrazy následně mohou být uloženy v Docker Registries. Ten slouží jako repositář pro obrazy, který může být buď veřejný, nebo soukromý. Veřejný repositář se nazývá Docker Hub [19], kde každý může získat kterýkoliv již vytvořený veřejný image, anebo může také nahrát vlastní. [17]

Docker v základu nabízí několik druhů síťových ovladačů, s jejichž pomocí lze efektivně propojit kontejnery mezi sebou. Docker zároveň podporuje implementaci vlastního síťového ovladače. [20]

K definici prostředí Docker podporuje dva způsoby. Prvním způsobem je tzv. *Dockerfile*. Dockerfile je soubor, který obsahuje všechny příkazy k vytvoření obrazu. Z takto definovaného obrazu lze následně vytvářet jednotlivé instance kontejnerů. [21]

Druhým způsobem je nástroj *Docker Compose*. Docker Compose umožňuje vytvořit multi-kontejnerové prostředí za pomocí definice dle formátovacího jazyku YAML. V něm lze následně definovat jednotlivé kontejnery a propojení mezi nimi. Díky tomuto přístupu je možné spouštět celé prostředí za pomoci jednoho příkazu. [22]

2.5.4 Podman

Podman je velice podobný předchozímu řešení Docker. Tento nástroj je primárně zaměřený na operační systém Linux, kde pomáhá najít, spustit a nasadit kontejnery dle pravidel OCI. V použití je dokonce až tak podobný, že většina uživatelů může změnit příkaz `docker` na `podman` bez problémů. Rozdíly tam ale přeci jen jsou. [23][24]

Podman oproti Docker používá takzvanou *daemon-less* architekturu. Jak už bylo zmíněno, Docker obsahuje neustále běžící proces, který se nazývá *daemon*, jenž poslouchá příkazy od klienta a spravuje všechny procesy spojené s kontejnerizací. Tento přístup vytváří problémy hlavně z bezpečnostního hlediska, jelikož daemon musí být spuštěn s právy správce počítače. Docker sice podporuje od verze v19.03 tzv. *rootless execution*, tedy mód, kdy nepotřebuje práva správce, ale při chodu v tomto módu mohou uživatelé narazit na různé limitace. [24]

Přesně na tento problém se zaměřil Podman. Podman používá pro správu kontejnerů nástroj *systemd*, který ke spuštění využívá práva uživatele, který provádí příkazy. Podman také využívá jiný nástroj *Buildah*, s jehož pomocí je schopen sestavovat kontejnery bez služby daemon. [24]

Jak bylo řečeno na začátku, Podman je nástroj primárně zaměřený na operační systém Linux, ovšem má i svou distribuci pro Windows. Ovšem k tomu, aby mohl na Windows fungovat, musí každý kontejner obsahovat hostující systém Linux, na kterém jsou následně kontejnery spuštěny. [23][24]

Kapitola 3

Analytická část

Tato kapitola se věnuje analýze aktuálního stavu testovací knihovny a možnostech jejího rozšíření.

3.1 Analýza stávajícího stavu knihovny

Původní testovací knihovna byla výstupem mé bakalářské práce [25] dokončené v roce 2021. Cílem této knihovny byla automatizace testů verifikace průmyslové komunikace a integrace do kontinuálního testování na Azure DevOps serveru. Vytvořená knihovna rozlišuje tři druhy účastníků testování:

Testovací služba Služba, která řídí testovací běh.

Testované zařízení Hlavní účastník testování, který běží na jiném zařízení, než ze kterého běží testovací služba.

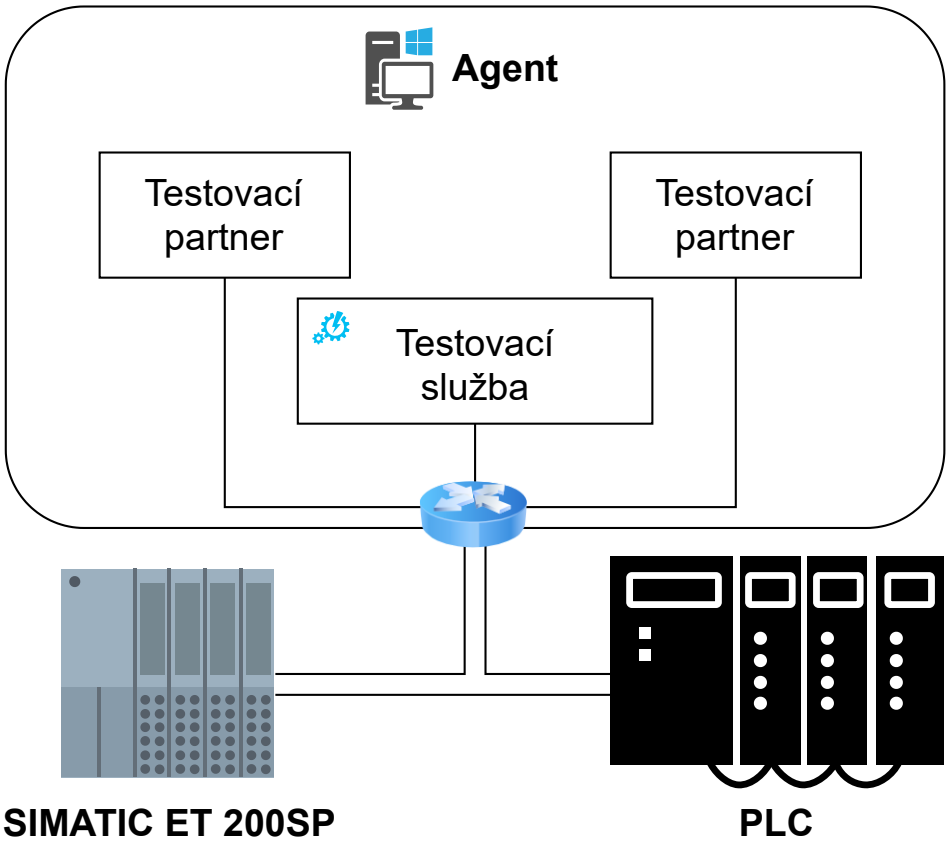
Testovací partner Zařízení, které simuluje nějaké testované zařízení. Toto zařízení běží na stejném zařízení, jako testovací služba.

Ideou knihovny je, že testovací služba, která řídí a synchronizuje běh na všech zúčastněných zařízeních, je spouštěna automatizovaně za pomoci Azure DevOps serveru v Azure Pipelines. Společně s ním je spuštěn i vyvíjený produkt, který je hlavním cílem testování, v tomto kontextu nazýván jako testované zařízení. Testované zařízení se následně připojí k testovací službě a po úspěchu této fáze započne samotné testování. Implicitně knihovna tedy vyžaduje alespoň jedno reálné zařízení.

Pro každý test lze definovat další účastníky testování - testovací partnery. Tito testovací partneři běží pouze po dobu daného testu a po dokončení testu zaniknou. Hlavní úlohou těchto zařízení je simulace protistrany při komunikaci.

Ukázka možného propojení všech účastníků testování je k vidění na obrázku 3.1. Jak je vidět, testovací partneři a testovací služba běží na jednom

zařízení, takzvaném agentovi, na kterém je primárně spouštěno celé testování. Vlevo dole je vidět testované zařízení, v tomto případě SIMATIC ET 200SP, které je propojeno s agentem. Vpravo dole můžeme vidět PLC, které pouze znázorňuje možnost propojení s dalšími externími zařízeními.



Obrázek 3.1: Ukázka možného propojení účastníků testování v původní knihovně. Zdroj Martin Štěpánek [25]

3.1.1 Integrace do testovaného zařízení

K tomu, aby mohla být knihovna použita na zařízení, které je testováno, je zapotřebí nejdříve implementovat rozhraní pro testované zařízení. Implementací rozhraní jsou definovány primárně všechny potřebné funkce pro vytvoření spojení mezi testovaným zařízením a testovací službou. Zároveň definujeme funkci pro získávání instancí jednotlivých testů.

Tyto testy rovněž dodržují jednotnou podobu pomocí rozhraní testu. Toto rozhraní definuje tři fáze testu:

1. Příprava na testování – definování potřebných struktur, inicializace.

2. Testování – provedení samotného testu.

3. Úklid po testu – uvolnění využitých zdrojů a uvedení zařízení do původního stavu.

Všichni účastníci testu musí pro provedení daného testu obsahovat implementaci daného testu, definovanou rozhraním pro test, a musí být na základě obdržení identifikátoru testu schopni test instanciovat. To se provede registrací testu ve funkci `getTest`, která je definována rozhraním pro zařízení.

Testovací služba následně synchronizuje všechny účastníky, tak aby před započetím další testovací fáze všechna zařízení dokončila fázi předchozí.

3.1.2 Virtualizační možnosti knihovny

Testovací knihovna aktuálně nepodporuje virtualizaci, jež byla definováno v sekci 2.1. Všechna zařízení, která byla vytvořena testovací knihovnou, běžela na stejném operačním systému bez žádného oddělení.

Testovací knihovna ovšem podporuje tzv. testovací partnery. Tato zařízení slouží k simulaci protistrany při testovaní komunikace s testovaným produktem. Testovací partneři byli vytvořeni tak, aby běželi nezávisle od testovací služby na vlastním vlákně. Implementace testovacího partnera pak už pouze požadovala dodání implementace testu, jelikož logika běhu zařízení byla už v testovací knihovně obsažena.

Každý testovací partner má životnost pouze v průběhu testu. To znamená, že před započetím testu je každý testovací partner vytvořen a připojen k testovací službě a následně po dokončení testu ukončen. Toto umožňuje variabilní počet testovacích partnerů pro každý test.

3.1.3 Topologie zapojení účastníků testu

V případě verifikace průmyslové komunikace je samozřejmě podstatné, aby všichni účastníci testu byli schopni komunikovat mezi sebou dle potřeb daných testů. Aktuální knihovna toto ale nijak nekontroluje.

Jediným požadavkem knihovny je, aby každý účastník testu byl připojen k testovací službě a odpovídal na její požadavky. Samotné testované zařízení se připojuje před započetím testování a do konce všech testů musí zůstat připojeno. Testovací partneři, jak už jsem zmínil, se připojují dle potřeby před započetím jednotlivých testů.

Knihovna tedy nijak neřeší topologii zapojení jednotlivých zařízení. Je zde tedy předpoklad, že testovací partneři budou schopni komunikovat s ostatními účastníky testu díky tomu, že samotná testovací služba je schopna s nimi komunikovat. Zároveň, pokud by z nějakého důvodu komunikace nebyla možná, tak musí selhat samotné testy.

3.1.4 Souhrn

I když testovací knihovna přináší zjednodušení a zautomatizovaní testování průmyslové komunikace, stále je zde prostor pro zlepšení. Knihovna v aktuální podobě podporuje pouze limitovanou virtualizaci. Za pomoci knihovny jsme sice schopni simulovat protistranu, ale již ne samotné zařízení, a to vždy musí běžet nezávisle na knihovně.

Aktuální požadavek na řešení topologie zapojení zařízení nesplňuje vůbec. Zároveň v již existujícím řešení limitující požadavek, že pro každý test musí každý účastník testu obsahovat implementaci testu. Toto nemusí být vždy potřeba, například při testování veřejného rozhraní.

3.2 Možnosti rozšíření virtualizačních prvků

Z analýzy stávajícího stavu knihovny je vidět, že možnost virtualizace prostřednictvím stávající knihovny jsou limitované. Knihovna nepodporuje možnost simulovat jakékoliv zařízení a vůbec neřeší topologii zapojení daných zařízení. Rozšířením knihovny tedy chceme docílit větší možnosti simulace jednotlivých zařízení a jejich propojení, čímž se zlepší simulace reálného prostředí. Nová knihovna by tedy měla cílit na

- možnost spustit jakékoliv zařízení jako virtualizované,
- a zapojit tato zařízení dle definované topologie.

Hlavní motivací je zvýšení flexibility knihovny, rozšíření možností využití a minimalizace úsilí potřebného na testovaní, což vede ke snížení nákladů na testování. Tyto kroky také směřují ke snížení závislosti na reálném hardwaru, na kterém běží účastníci testů. Je ale samozřejmé, že testy na reálném hardwaru budou vždy potřeba a nelze se této závislosti nijak zbavit.

3.2.1 Virtualizace zařízení

Možností jak virtualizovat zařízení existuje několik. Tyto techniky byly přiblíženy v sekci 2.1. Co je tedy nejvhodnějším přístupem k virtualizaci námi požadovaných průmyslových zařízení?

Jednou z možností je použít QEMU, které dokáže emulovat širokou škálu platforem. Díky emulaci by teoreticky mělo být možné emulovat jakékoliv zařízení s minimálním potřebou zásahu do kódu daného zařízení.

Další možností je vytvoření VM pro každé virtualizované zařízení. Volba operačního systému poté závisí na zařízení. Pro zachování univerzálnosti hostitelského zařízení dává smysl použití hypervizoru typu II.

Logickou možností je také použití virtualizace skrze kontejnerizaci. Díky kontejnerizaci nemusíme virtualizovat celé operační systémy, ale pouze jejich části, čímž můžeme ušetřit výpočetní zdroje.

Který přístup je tedy efektivnější? Dle studií, jež zkoumaly rozdíl ve výkonu a efektivnosti využití výpočetních zdrojů mezi kontejnerizací prostřednictvím softwaru Docker a virtualizačního řešení KVM, jenž je využíván i QEMU [26], který umožňuje jádru fungovat jako hypervizor, je lepším přístupem kontejnerizace.

Při jejich porovnání bylo zjištěno, že s použitím kontejnerizačního řešení Docker jsou virtualizovaná zařízení spouštěná výrazně rychleji. Tato zařízení také spotřebovávají méně výpočetních zdrojů. V porovnání při počítání HPC(High performance counting) úkolů, bylo dosaženo díky nástroji Docker lepšího výkonu o 42 % při úkolech náročných na procesor a o 14,98 % lepšího výkonu při úkolech náročných na interní paměť. [27][28]

Jak je tedy vidět, virtualizace kontejnerizací je efektivnější. V následujících sekcích tedy budeme uvažovat pouze nad virtualizací za pomoci kontejnerizace.

3.2.2 Porovnání kontejnerizačních řešení

V dnešní době existuje několik řešení, která využívají kontejnerizaci. V následující sekci budou porovnána jednotlivá vybraná řešení a zároveň bude zhodnocena jejich vhodnost pro použití v testovací knihovně.

Primárním cílem použít k řešení kontejnerizace tzv. *container engine*. Jeho přímým využitím, oproti využití několika nástrojů, které by dohromady tvořily container engine, se, doufám, předejde problémům s kompatibilitou a integrací těchto nástrojů dohromady. Nic ovšem nebrání využití řešení, které už všechny použité nástroje integrovalo dohromady.

Z technologického hlediska jediným požadavkem na dané řešení je podpora operačního systému Windows. Tento požadavek primárně vyvstává z infrastruktury firmy Siemens.

Vybranými řešeními tedy jsou

- Docker,
- Podman.

Obě tato řešení jsou srovnatelná. Jak bylo zmíněno, ve spoustě případů je možné zaměnit příkaz `docker` za `podman` bez jakýchkoli problémů.

Pokud se ovšem zaměříme na jejich rozdíly, tak Docker v základu podporuje více síťových ovladačů. Je však možné také implementovat vlastní síťový ovladač. [20]

Největší výhoda ovšem spočívá v rozšířenosti softwaru Docker. Při pohledu na dostupnost knihoven integrujících software Docker do prostředí .NET, ve kterém běží stávající knihovna, je vidět rovnou několik možností, které lze využít. Oproti tomu pro software Podman tyto integrace neexistují. Při jeho integraci by tedy bylo potřeba vytvořit vlastní integraci.

Zároveň je Podman navrhnut primárně pro operační systém Linux. Jeho použití je sice na jiném operačním systému možné, ale každý kontejner musí obsahovat hostující Linuxový systém.

Jak je tedy vidět, pro použití v testovací knihovně je vhodnější využití softwaru Docker. V aktuální podobě jeho použití přináší jasné benefity.

3.3 Možnost propojení s reálnými zařízeními

Testovací knihovna ve své aktuální podobě podporuje propojení s reálnými zařízeními, na kterých běží implementace propojení s testovací službou. Propojení mezi zařízeními samotnými je teoreticky také možné, ovšem už není nijak kontrolováno či orchestrováno za pomoci testovací knihovny.

Vyvstává zde tedy otázka, zdali je teoreticky možné propojit jednotlivá virtualizovaná zařízení s reálnými zařízeními, která běží mimo zařízení, na kterém běží virtualizované prostředí. K tomu, aby bylo možné toto analyzovat, předpokládejme dále využití softwaru Docker pro virtualizaci v testovací knihovně.

3.3.1 Analýza

Jednotlivé Docker kontejnery nemají ponětí o propojení s externí sítí, tedy sítí mimo virtualizované prostředí. Kontejnery, pokud jsou připojeny k nějaké síti, mají pouze znalost o implicitní bráně, na kterou směřují veškerou komunikaci. Pokud kontejnery nejsou izolované, jsou takto schopné dosáhnout hostujícího stroje a tím i externí sítě. V komunikaci ve směru od virtualizovaného zařízení k reálnému zařízení by tedy neměl být problém. [29]

Problém může ovšem nastat v opačném směru. Jednotlivé kontejnery v základním nastavení nejsou dostupné z externí sítě. Docker ovšem podporuje přesměrování portů. Tímto způsobem je možné namapovat port z virtualizovaného prostředí na port na hostujícím zařízení, na který je následně směřována komunikace z externího zařízení. [30]

Vyvstává zde však problém vznikající primárně z definice jednotlivých komunikačních protokolů. Tyto protokoly často běží na specifickém portu a tedy komunikace s více než jedním zařízením může způsobit kolizi těchto portů. To může zapříčinit neschopnost komunikace se všemi virtualizovanými zařízeními.

3.3.2 Shrnutí

Jak z analýzy vyplývá, komunikace s reálnými zařízení je teoreticky možná. Ze specifika daného problému a protokolů využitých ke komunikaci ovšem nebude možné při využití kontejnerizace s pomocí softwaru Docker komunikovat s externími zařízeními. K tomu, aby byla komunikace možná, je potřeba nejprve vyřešit problém kolize portů, pokud tedy nějaké řešení pro tento problém existuje.

Kapitola 4

Návrh

Tato kapitola se zabývá návrhem úprav a rozšíření původní testovací knihovny.

4.1 Návrh změny architektury

Jednu z možných architektur stávající testovací knihovny bylo možné vidět na obrázku 3.1. Předchozí přístup vyžadoval existenci alespoň jednoho fyzického zařízení, tedy reálného zařízení běžícího mimo zařízení, na kterém byla spouštěna testovací knihovna. To byla velice omezující podmínka, kvůli které nebylo možné virtualizovat samotné testované zařízení.

Představu nové architektury lze spatřit na obrázku 4.1. Jak je na obrázku vidět, vše se již odehrává na tzv. agentovi, jímž může být jakékoliv zařízení s potřebným softwarem pro virtualizaci. Všechna zařízení, mimo testovací služby, jsou přesunuta do virtualizovaného prostředí, které bude orchestrováno za pomoci softwaru Docker.

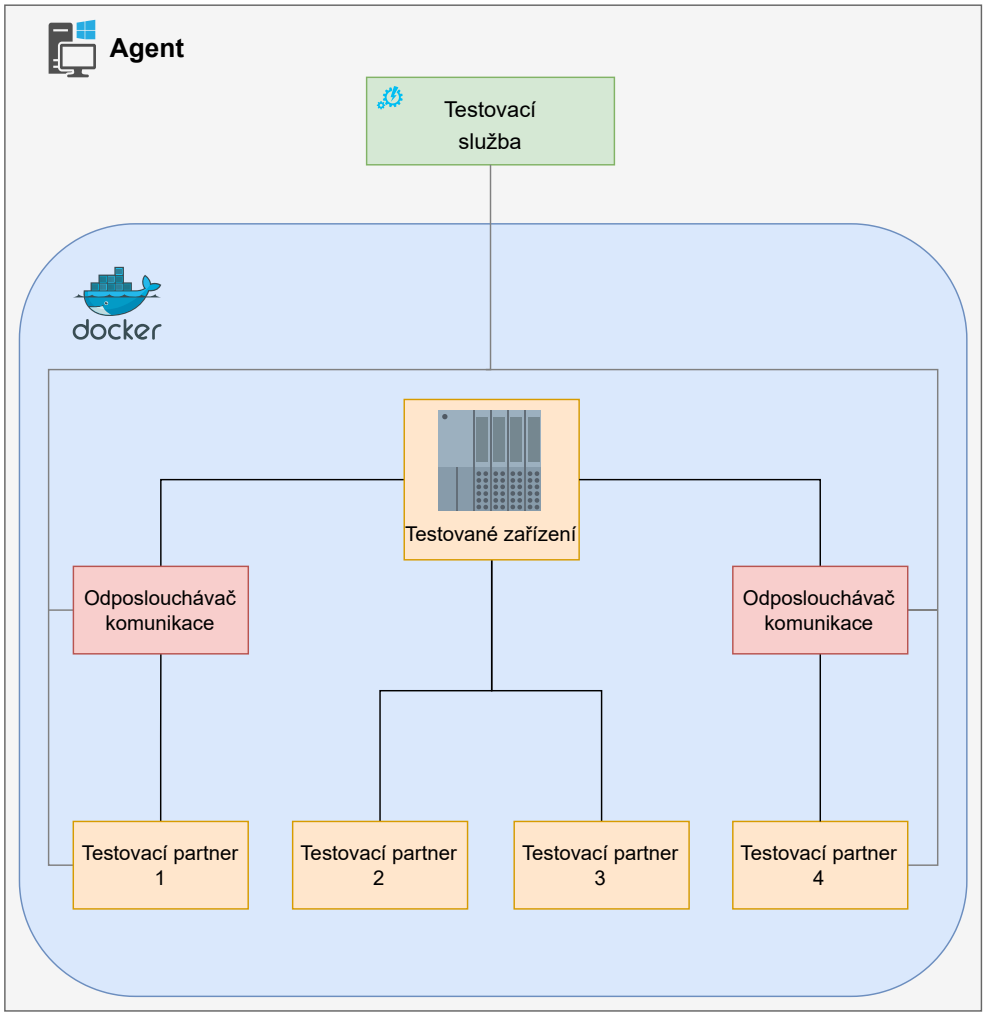
Na obrázku lze vidět tři druhy zařízení, která jsou barevně oddělená. Zeleně označeným zařízením je původní *testovací služba*. Ta řídí testovací běh a vyhodnocuje výsledky testů. Testovací služba běží přímo v operačním systému agenta, tedy bez žádné virtualizace.

Uvnitř modře označeného nového virtualizovaného prostředí lze následně vidět jednu z možných simulovaných topologií, v tomto případě topologii hvězdy. Každý objekt uvnitř virtualizovaného prostředí představuje právě jedno zařízení, které je kontejnerizováno. Všechna oranžově označená virtualizovaná zařízení představují účastníky testu, tedy zařízení, která určitým způsobem přispívají k provedení daného testu. *Testované zařízení* představuje vyvíjený produkt, který chceme testovat. Ostatní oranžová zařízení představují *testovací partnery*. Ti přispívají k vytvoření prostředí testu a k orchestraci testu.

Rozdílem od stávající knihovny je, že ne všichni účastníci testu musí být k testovací službě připojeni. Na obrázku lze vidět, že v tomto případě jsou pouze

testovací partner 1 a 4 připojeni k testovací službě. Je zde ale předpoklad, že alespoň jedno zařízení musí být k testovací službě připojeno.

Novým typem zařízení je červeně označený *odposlouchávač komunikace*. Toto zařízení, jak už z názvu vyplývá, bude zaznamenávat komunikaci na právě jednom spojení dvou zařízení. Zároveň bude připojeno k testovací službě. Díky tomuto spojení bude možné zaznamenávat komunikaci pouze v průběhu testu, což ve výsledku bude znamenat, že pro každý test bude existovat separátní záznam komunikace.



Obrázek 4.1: Ukázka jedné z možných architektur nové testovací knihovny

4.1.1 Propojení s externími komponentami

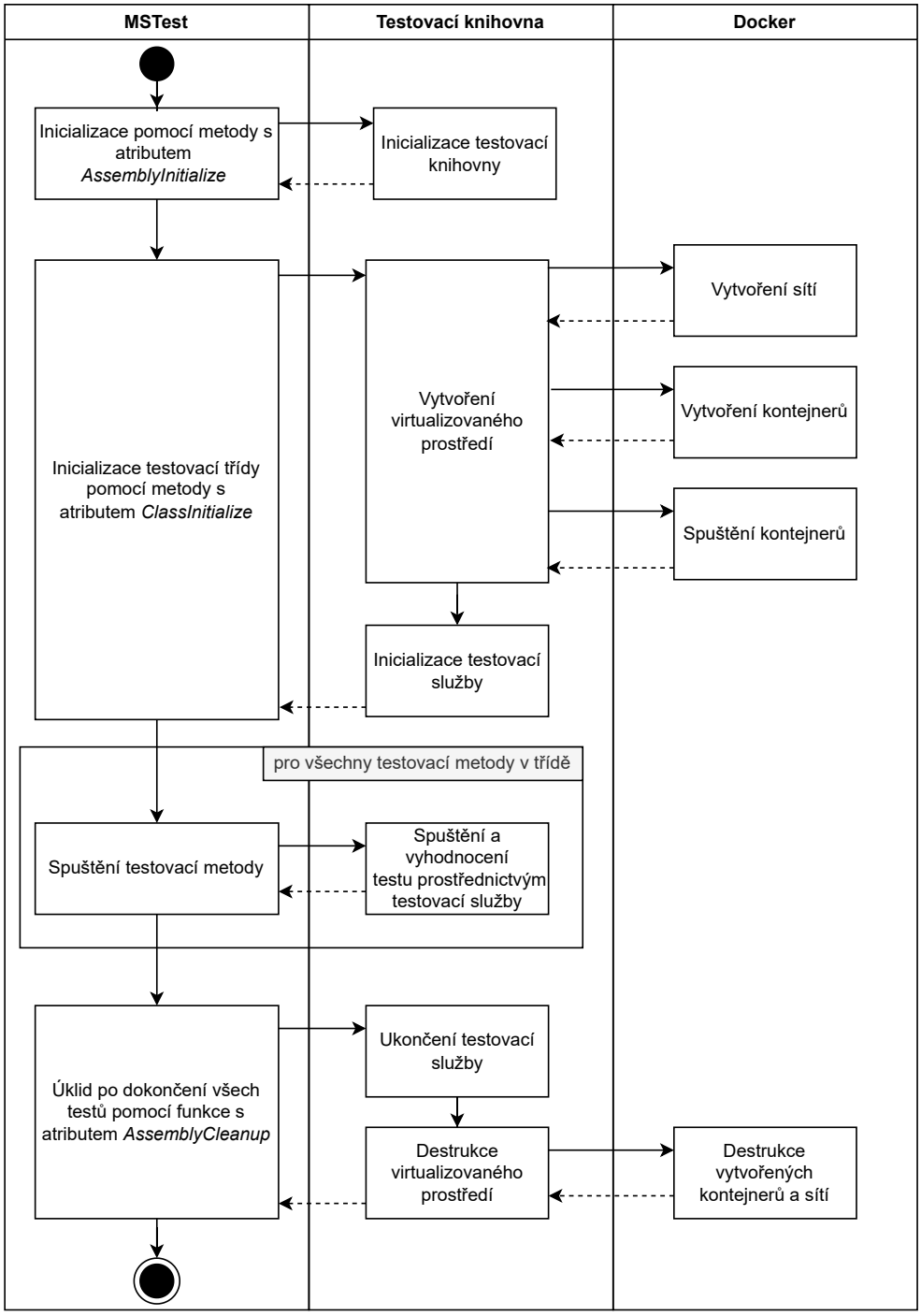
Testovací knihovna bude využívat k testování tyto komponenty:

- Testovací knihovnu MSTest
- Kontejnerizační řešení Docker

Každá komponenta má své důležité využití. Testovací knihovna MSTest, která je již využita ve stávající knihovně, umožňuje testy spouštět díky její široké integraci. S její pomocí je mimo jiné možné testy spouštět na Azure DevOps, díky čemuž lze dosáhnout plné automatizace testování.

S pomocí virtualizačního řešení Docker bude knihovna následně schopna vytvářet jednotlivá virtualizovaná zařízení a následně je propojovat mezi sebou do požadovaných topologií.

To vše umožní testovací knihovně úspěšně testovat. Jejich propojení můžeme vidět na obrázku 4.2. Jak lze vidět, testovací knihovna je ovládána prostřednictvím testovací knihovny MSTest. Ta poté provede všechny potřebné operace, jako vytvoření testovací služby a vytvoření virtualizovaného prostředí, které je vytvářeno pomocí softwaru Docker. Všechny vytvořené zdroje jsou poté na konci testování odstraněny.



Obrázek 4.2: Propojení a komunikace mezi komponenty v testovací knihovně

4.2 Orchestrace virtualizovaného prostředí

Důležitou součástí testovací knihovny bude implementace orchestrace virtualizovaného prostředí. To znamená, že testovací služba bude schopna vytvářet jednotlivá virtuální zařízení, které následně automaticky propojí dle požadované topologie a po konci testování daná zařízení ukončí.

Knihovna bude obsahovat tzv. *správce virtualizovaného prostředí*. Ten se bude starat o všechny úkony spojené s vytvářením a chodem virtualizovaného prostředí. Tedy tento správce bude vytvářet potřebné kontejnery a sítě, s jejichž pomocí propojí všechna zařízení dle konfigurace.

4.2.1 Nastavení virtualizovaného prostředí

Virtualizované prostředí bude definované za pomoci konfiguračního souboru dle formátu YAML. Hlavní důvod, proč využít YAML namísto původního formátu JSON, kterým bylo definováno nastavení testovací knihovny, je co nejbližší přiblížení se aktuálním konvencím v kontextu k Docker compose. Konfigurace bude tedy co nejvíce napodobovat jeho konfigurační soubor.

První položkou v konfiguračním souboru bude položka `label`. Ta bude obsahovat označení daného virtualizovaného prostředí. Další položkou v konfiguračním souboru bude položka `service`. Ta bude obsahovat pouze položku `connections`, která bude obsahovat počet zařízení, jež se připojí k testovací službě. Toto číslo musí být větší než 0.

Další položkou bude položka `containers`. Ta bude obsahovat informace o všech zařízeních, která v daném virtualizovaném prostředí budou. Klíčem každé položky uvnitř této sekce bude název zařízení, které bude fungovat pro referenci daného zařízení. Pro každé zařízení následně půjde definovat tato nastavení:

- `build` - Definice zařízení, buď za pomoci obrazu (image), nebo cestou k docker souboru (Dockerfile). Toto bude označeno klíčem `image` a `dockerfile`, následovanou požadovanou hodnotou.
- `cap_add` - Seznam „schopností“, v tomto kontextu práv, které bude daný účet na kontejneru mít. Ty budou přidány k právům, která v základu docker dává účtu vytvořenému v kontejneru.
- `commands` - Seznam příkazů, které budou spuštěny v kontejneru.
- `environment` - Seznam proměnných, které budou v kontejneru nastaveny. Klíč bude název proměnné a jeho hodnota bude hodnotou proměnné
- `ip_mapping` - Seznam mapování IP adres na domény. Klíčem je název domény a hodnotou je IP adresa, na kterou má daná doména ukazovat.

- `ports` - Seznam mapování portů z kontejneru do prostředí hostitelského zařízení. Každý záznam obsahuje dva porty oddělené dvojtečkou, kdy první je port hostitelského zařízení a druhý je port kontejnerizovaného zařízení.

- `privileged` - Boolean hodnota, která při hodnotě `true` přiřadí účtu správcovská práva

- `logging` - Boolean hodnota, která značí, zdali je komunikace mezi tímto zařízením odposlouchávána. Pokud je hodnota `true`, tak poté knihovna zařídí zaznamenávání všech spojů tohoto zařízení.

- `type` - Typ zařízení, které bude vytvořeno. Typy zařízení jsou blíže vysvětleny v sekci 4.2.2.

- `tty` - Boolean hodnota, která v původní definici značí připojení pseudoterminálu ke kontejneru, což má za následek to, že kontejner po dokončení definovaných příkazů zůstane běžet. V kontextu knihovny to bude znamenat, že kontejner zůstane po dokončení všech příkazů definovaných v `commands` běžet.

V neposlední řadě bude konfigurační soubor obsahovat položku `links`. Ta bude definovat jednotlivá propojení mezi zařízeními, tedy celkovou topologii. V tomto seznamu bude každá položka obsahovat názvy dvou zařízení oddělené dvojtečkou a pro každý záznam bude vytvořeno propojení mezi danými zařízeními.

Povinnou součástí konfiguračního souboru bude celá sekce `service`, sekce `containers`, kde musí být definováno nejméně jedno zařízení, které musí obsahovat alespoň kolonky `build` a `type`. Ostatní budou nepovinné. Platí zde však podmínka, že každé zařízení musí dosáhnout všech ostatních zařízení ve virtualizovaném prostředí.

Testovací knihovna bude kontrolovat, zdali testované prostředí bylo vytvořeno. Pokud ne, nezapočne testování. Zároveň bude kontrolovat zda všechna zařízení byla úspěšně spuštěna. Po konci testování testovací knihovna všechna zařízení ukončí a vymaže všechny jím definované sítě, kontejnery atd.

Pro pohodlnost bude testovací knihovna také podporovat více různých topologií v rámci jednoho testovacího projektu. Při každé inicializaci virtualizovaného prostředí testovací služba porovná danou konfiguraci s aktuální konfigurací. Pokud budou konfigurace totožné, testovací služba ponechává prostředí nepozměněné. V opačném případě vyčistí virtualizovaný prostor, do kterého následně vloží nová zařízení a jejich nastavení.

4.2.2 Definice zařízení

Testovací knihovna bude pro definici kontejnerizovaného zařízení obsahovat rozhraní pod názvem `IVMContainer`. Toto rozhraní bude obsahovat následující metody:

- `AddCapability(string)` - přidá schopnost, neboli práva, kontejneru.
- `AddCommand(string)` - přidá příkaz, který bude spuštěn po spuštění kontejneru.
- `AddHostIpMapping(string, string)` - přidá mapování domény na IP adresu. Prvním argument je název domény, druhým IP adresa dané domény.
- `Build()` - sestaví daný kontejner.
- `Dispose()` - zničí všechny alokované prostředky, včetně vytvořeného kontejneru v Docker.
- `ExposePort(int,int)` - zpřístupní port kontejneru do hostitelského zařízení. Prvním argumentem je číslo portu v kontejneru, druhým argumentem je port, ze kterého daný port bude přístupný na hostitelském zařízení.
- `GetLog()` - získá z kontejneru všechny zápisy, které v něm byly vypsány.
- `IsRunning` - vlastnost, která značí, zda daný kontejner je spuštěn.
- `Mount(string,string)` - zpřístupní složku na hostitelském zařízení v kontejneru. Prvním argumentem je cesta na hostitelském zařízení, druhým cesta v kontejneru.
- `RunPriviledged()` - dá správcovská práva kontejneru.
- `SetEnvironmentVariables(Dictionary<string, string>)` - přidá do prostředí kontejneru proměnné, definované v předaném slovníku.
- `SetNetworkConf(ContainerNetworkConf)` - přidá nastavení sítě.
- `Start()` - spustí daný kontejner. Metoda vrací boolean hodnotu o správnosti spuštění kontejneru.
- `Stop()` - zastaví daný kontejner.
- `Tty()` - donutí kontejner běžet i po dokončení všech příkazů.

Správce virtualizovaného prostředí následně bude schopen nastavit daná zařízení dle implementace předem definovaného rozhraní. Druh implementace bude určen položkou `type` v konfiguraci zařízení. Každá implementace daného rozhraní musí zajistit správné nastavení všech konfiguračních položek definovaných rozhraním kontejnerizovaného zařízení.

4.3 Komunikace

Komunikaci v rámci testovací knihovny lze rozdělit do dvou kategorií:

1. komunikace mezi zařízeními
2. komunikace s testovací službou

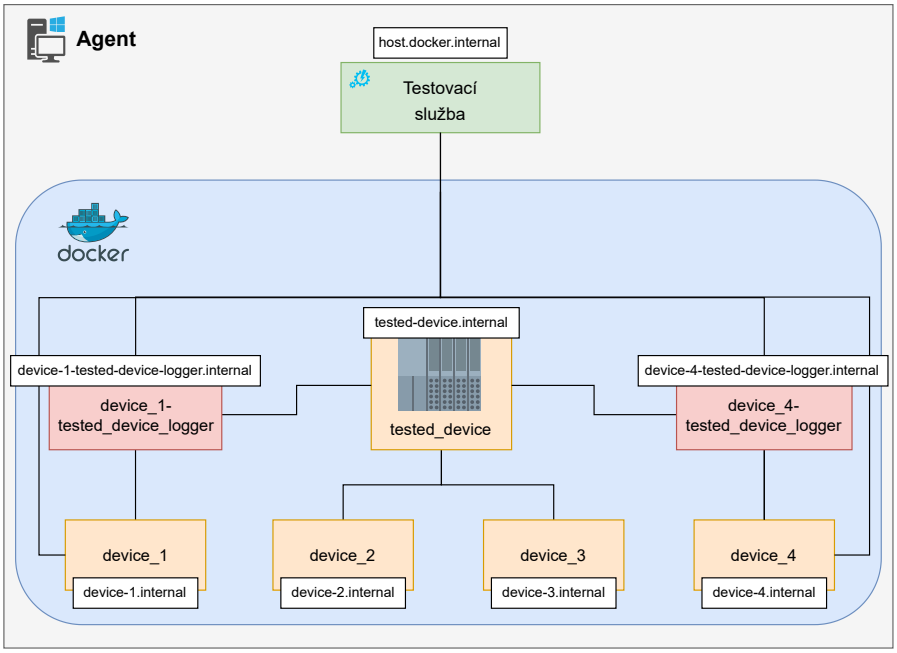
Veškerá komunikace mezi zařízeními bude probíhat uvnitř virtualizovaného prostředí. Jejich propojení bude zajištěno pomocí testovací knihovny. Ta dle konfigurace, definované v sekci 4.2.1, vytvoří jednotlivá propojení mezi zařízeními s pomocí softwaru Docker. Tato komunikace nebude nijak ze strany testovací knihovny kontrolována, pouze může být v průběhu testu zaznamenána.

Testovací knihovna pro každé zařízení nastaví seznam interních domén, skrz které bude možné adresovat účastníky testu. Tyto domény budou ve tvaru `[jméno-zařízení].internal`. Validními znaky pro domény budou pouze písmena, čísla a pomlčka. Jméno zařízení tedy bude transformováno tak, aby obsahovalo pouze tyto znaky. Všechny nepodporované znaky budou z jména zařízení odstraněny. Výjimkou je pouze znak `'_'`, který bude nahrazen pomlčkou. V případě kolize dvou jmen po transformaci bude do jména domény přidán index.

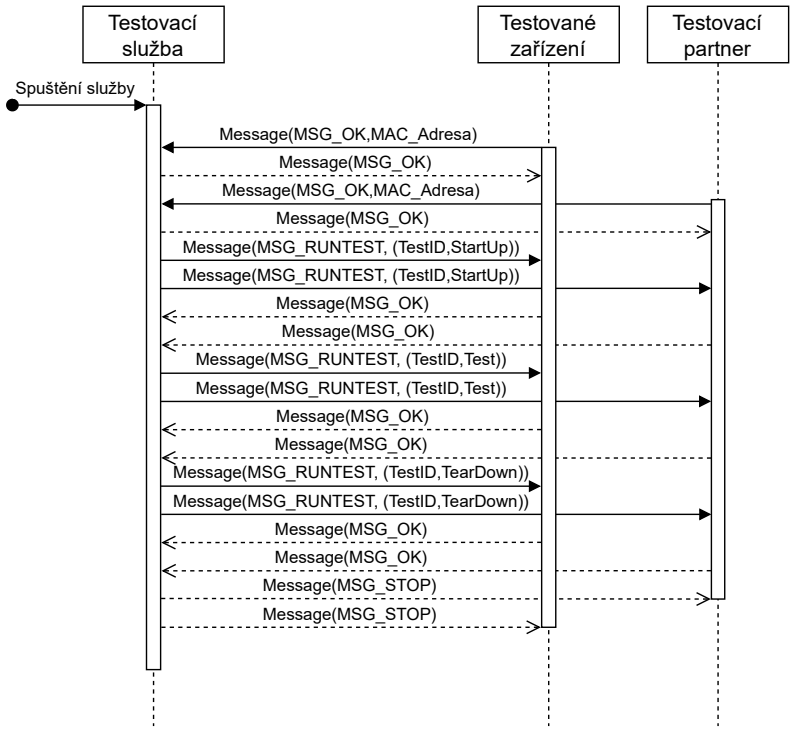
Ukázku přiřazení domén můžeme vidět na obrázku 4.3. Jak je vidět, topologie je totožná s ukázkovou topologií na obrázku 4.1, pouze ke všem zařízením uvnitř virtuálního prostředí bylo přiřazeno unikátní jméno. Každé zařízení má přiřazenou doménu dle daného formátu. Výjimkou je pouze testovací služba, která je mimo virtualizované prostředí. Software Docker automaticky definuje doménu `host.docker.internal`, s pomocí které je možné adresovat hostující stroj a tím pádem i testovací službu.

Komunikace s testovací službou bude probíhat za stejných podmínek, jako tomu bylo doposud. Testovací knihovna přesně definuje strukturu zpráv. Každá zpráva obsahuje v prvních dvou bajtech typ zprávy, v dalších dvou bajtech délku dat zprávy a případná data zprávy, jejichž délka je uložena v délce dat zprávy. Každá zpráva tedy musí mít minimálně 4 bajty. Veškerá data zprávy jsou uložena dle formátu *big-endian*, respektive na paměťovém místě s nejnižší adresou je uložen nejvýznamnější bit.

Možnou komunikaci mezi testovací službou a účastníky testu, tedy zařízeními uvnitř virtualizovaného prostředí, můžeme spatřit na sekvenčním diagramu, který je na obrázku 4.4. Na něm lze vidět, jak by probíhala komunikace při běhu o jednom testu. Jak jsem již ale zmínil dříve, dle nového přístupu nemusí všechna zařízení komunikovat s testovací službou.



Obrázek 4.3: Ukázka přiřazení domén k zařízením

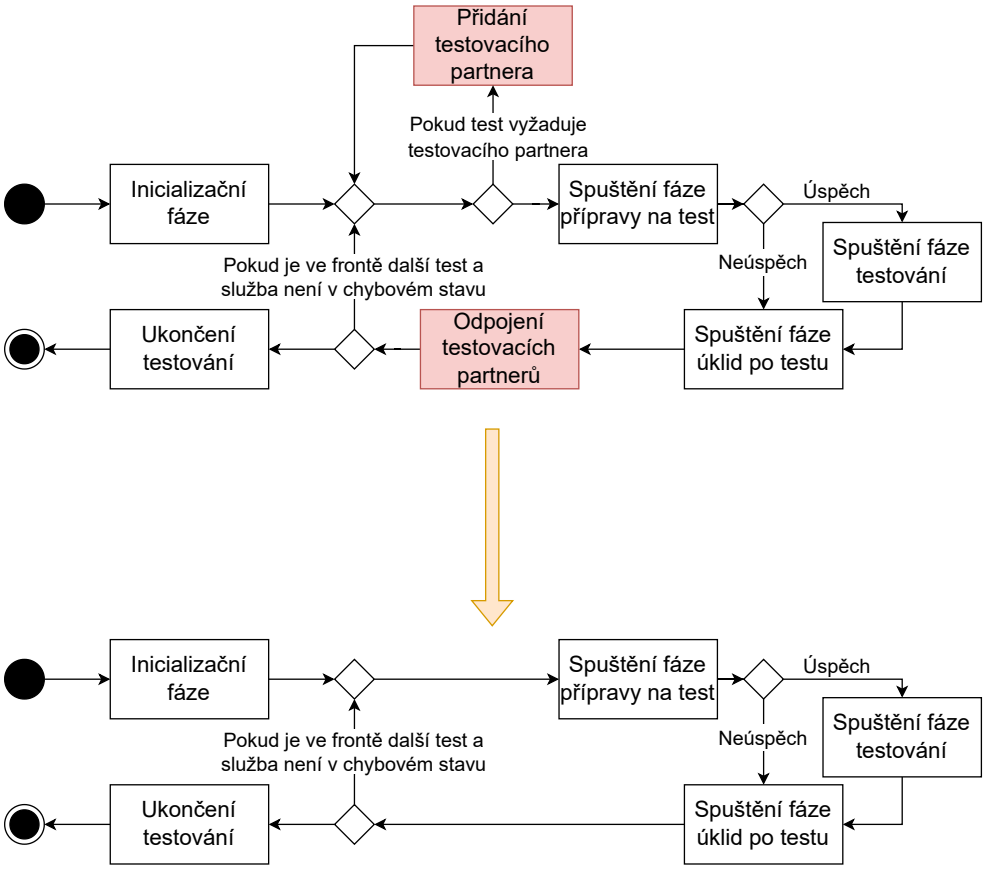


Obrázek 4.4: Sekvenční diagram ukázky komunikace mezi účastníky testování. Zdroj Martin Štěpánek [25]

4.4 Testovací služba

Jádrem testovacího běhu zůstane testovací služba, která musí komunikovat a tím ovládat alespoň jedno zařízení. Ve stávající implementaci byla na počátku testování vytvořena jedna instance testovací služby, která existovala po celou dobu testování. Nově bude životnost testovací služby navázána na dané virtualizované prostředí. Testovací služba tedy vznikne až po úspěšném vytvoření virtualizovaného prostředí. Při změně virtualizovaného prostředí bude testovací služba ukončena a po vytvoření nového prostředí spuštěna nová instance.

Fungování testovací služby lze spatřit na novém diagramu aktivit testovací služby na obrázku 4.5, na kterém jsou označeny i změny oproti původnímu diagramu aktivit testovací služby. Jak můžeme na diagramu vidět, nově již nelze přidávat nové testovací partnery po dokončení inicializační fáze testovací služby.



Obrázek 4.5: Nový diagram aktivit testovací služby. Podle Martina Štěpánka [25]

Po sestavení virtualizovaného prostředí testovací služba započne inicializační fázi, ve které se všechna zařízení ovládaná testovací službou připojí k

testovací službě. Tedy mimo přímých účastníků testu se v této fázi připojí i všichni odposlouchávači komunikace, pokud nějací existují.

Testovací služba sama o sobě nemá ponětí o tom, které testy budou spuštěny a které ne. Testovací služba skrz definované API definuje metodu, skrz kterou je možné spustit test dle identifikátoru. Testovací služba následně spouští všechny fáze testu. Nakonec služba při ukončení odesílá všem připojeným účastníkům zprávu o ukončení testování, čímž mohou být připojená zařízení taktéž ukončena. O úspěšné ukončení všech zařízení se ovšem stará správce virtualizovaného prostředí.

4.5 Testování

Nová testovací knihovna zachová proces testování a podobu jednotlivých testu ve stejné podobě, jako tomu bylo doposud. K umožnění testování knihovna definuje dvě rozhraní:

- Rozhraní pro účastníka testu (dříve pojmenováno jako Rozhraní pro testované zařízení)
- Rozhraní pro test

Rozhraní pro testované zařízení definovalo potřebné rozhraní pro vytvoření připojení s testovací službou a pro získání jednotlivých instancí testů. Následně testovací knihovna mohla za pomoci tohoto rozhraní vést testovací běh zařízení. Toto rozhraní lze ale využít pro jakékoliv zařízení, které je potřeba ovládat testovací službou. Z tohoto důvodu bylo tedy rozhraní přejmenována na *Rozhraní pro účastníka testu*.

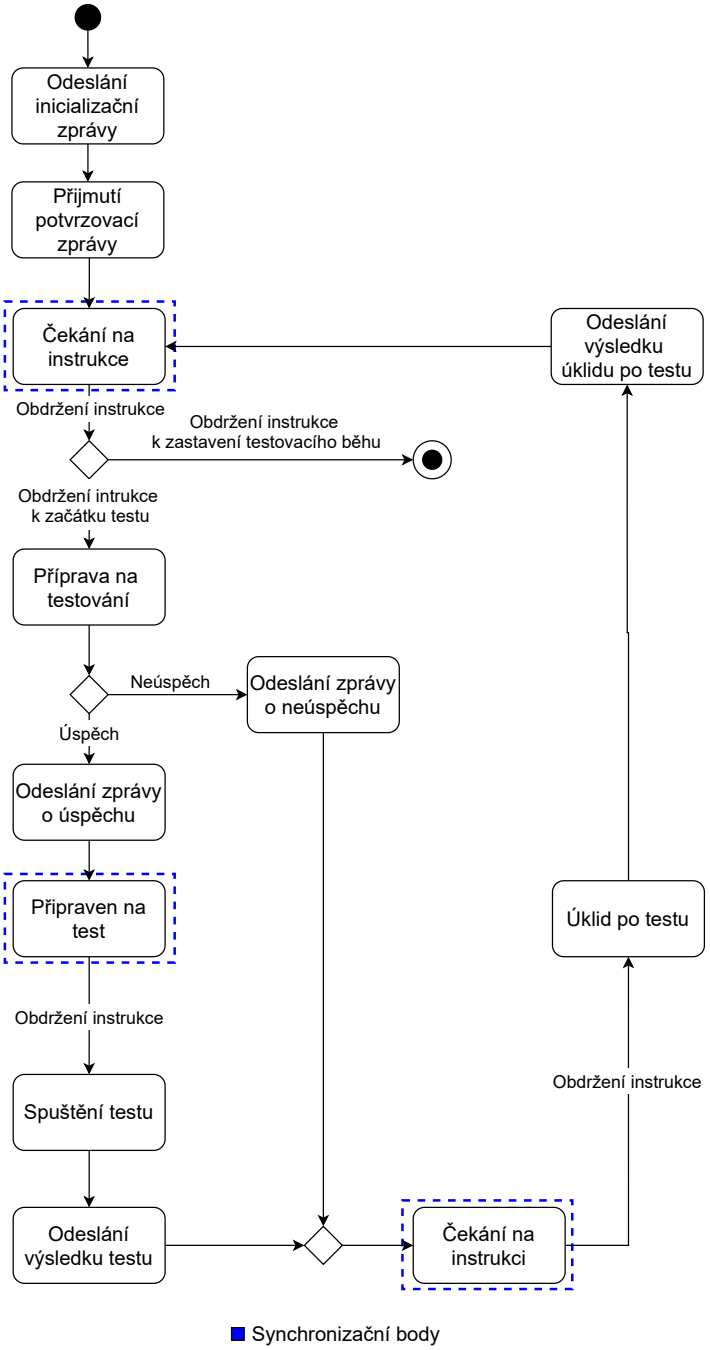
Rozhraní pro test zůstalo nepozměněné. Pro připomenutí, rozhraní definuje tři fáze testu, které jsou:

1. Příprava na testování – definování potřebných struktur, inicializace.
2. Testování – provedení samotného testu.
3. Úklid po testu – uvolnění využitých zdrojů a uvedení zařízení do původního stavu.

Běh účastníka testu, který je připojen k testovací službě, můžeme vidět na diagramu aktivit na obrázku 4.6. Jeho aktivity zůstanou identické. Změna nastává v definovaných synchronizačních bodech, které jsou na obrázku označeny modře.

Testovací služba obdrží informace o dokončení každé fáze testu pouze od těch účastníků, již jsou k ní připojeni. Z tohoto důvodu tedy není schopna synchronizovat ostatní zařízení. Pokud je zde tedy potřeba synchronizovat nějaké zařízení, které není připojeno k testovací službě, může tato synchronizace

probíhat skrz některého z účastníků testu, který k testovací službě připojen je. Implementace je ovšem už na samotném uživateli knihovny.



Obrázek 4.6: Diagram aktivit účastníka testu

Kapitola 5

Implementace

Tato kapitola popisuje implementaci navržených úprav z kapitoly 4.

5.1 Příprava stávající implementace testovací knihovny

Původní testovací knihovna obsahovala toto rozdělení implementace, dle zdrojových složek:

- `core` - implementace celé testovací knihovny v jazyce C#, tedy implementace testovací služby, testovacích partnerů atd.
- `cpp` - implementace pro účastníka testování v jazyce C++, včetně potřebných rozhraní.

V předchozí implementaci nebylo potřeba žádné zařízení vytvářet externě mimo testovací projekt. Proto veškerá implementace v jazyku C# byla v jednom celku. Ovšem účastník testu vůbec nepotřebuje implementaci testovací služby a integraci virtualizovaného prostředí. Proto tedy bylo logické rozdělit testovací knihovnu na tyto logické celky:

- `TestLib.Core` (složka `core`) - jádro testovací knihovny, které obsahuje všechna potřebná rozhraní, definice zpráv a správce běhu testovacího partnera, respektive účastníka testu.
- `TestLib` (složka `test_lib`) - celá testovací knihovna, která mimo jádra testovací služby obsahuje testovací službu, integraci virtualizovaného prostředí a další pomocné entity. Jádro testovací knihovny je přidáno jako závislost.

Jednotlivá zařízení jsou s využitím jádra testovací knihovny připojena k testovací službě, která následně může ovládat testovací běh na zařízení. Jádro

testovací knihovny je implementováno také v jazyce C++ které bylo součástí již přechozí testovací knihovny, a Python. Ta vznikla z důvodu pohodlnosti testerů, kteří píší velkou část testů právě v tomto jazyce. S její pomocí budou schopni jednodušeji integrovat stávající testy do nové testovací knihovny.

5.1.1 Úprava testovací služby

S příchodem nové konfigurace virtualizovaného prostředí obdržela i konfigurace testovací služby novou konfiguraci. Konfigurační soubor testovací služby je tedy možné nyní definovat taktéž ve formátu YAML namísto formátu JSON. Konfigurační soubor obsahuje tyto konfigurační možnosti:

- `ip` - IP, na kterém bude poslouchat testovací služba, výchozí hodnota je `127.0.0.1`
- `port` - port, na kterém bude poslouchat testovací služba, výchozí hodnota je `1337`

Tento konfigurační soubor již není potřeba definovat, pokud jsou výchozí hodnoty dostatečné. V opačném případě lze v konfiguračním souboru definovat pouze ty hodnoty, které je potřeba změnit.

S novým konfiguračním souborem byl změněn i způsob, jakým testovací služba obdrží tento konfigurační soubor. Nově testovací služba obdrží konfiguraci ve svém konstruktoru, místo toho, aby ho získávala ze statické třídy `Global`. Nově také ve svém konstruktoru obdrží konfiguraci virtualizovaného prostředí. Z této konfigurace následně získá počet očekáváných zařízení, která se k testovací službě připojí.

Testovací služba nově dle návrhu nemůže přidat žádné zařízení po dokončení inicializační fáze. Všechny metody, které toto umožňovaly, byly odstraněny. Jediné zařízení, které se nyní odlišuje od ostatních, je odposlouchávač komunikace. Ten má MAC adresu `CA:FE:C0:FF:EE:00`, se kterou se identifikuje v první zprávě a díky tomu je možné ho rozeznat od ostatních zařízení.

Toto odlišení ovšem způsobuje pouze jednu věc. Testovací služba nyní odesílá zprávu o započetí první fáze testu jako první odposlouchávačům komunikace. Naopak při poslední fázi testu je odposlouchávačům komunikace zpráva o započetí poslední fáze testu odeslána jako posledním. Je to z toho důvodu, aby odposlouchávače byly schopny zaznamenat veškerou komunikaci.

5.1.2 Komunikace

Menší změny nastaly i v případě komunikace mezi testovací službou a účastníky testu. Nyní jsou všechny zprávy odesílány podle formátu big-endian. O převod na endianitu platformy se následně stará implementace zprávy, obsažena v třídě `Message`.

Nově také všechny funkce/metody, které zajišťovaly příjem zpráv, obsahují argument velikosti očekávané zprávy. Díky tomu lze zprávy jednoduše oddělit a zpracovat je na správném místě bez nutnosti implementace vyrovnávací paměti.

5.2 Orchestrace virtualizovaného prostředí

Testovací knihovna k úspěšnému vytvoření potřebuje umět primárně tyto aktivity:

1. Umět komunikovat se softwarem Docker.
2. Vytvořit kontejnery, tedy virtualizovaná zařízení, dle konfigurace.
3. Propojit všechna zařízení mezi sebou, taktéž dle konfigurace.

5.2.1 Komunikace se softwarem Docker

Orchestrace virtualizovaného prostředí má dle návrhu probíhat s pomocí softwaru Docker. Jako první tedy bylo potřeba integrovat komunikaci se softwarem Docker do testovací knihovny. Před vytvořením vlastní integrace je ale rozumné se nejdříve porozhlédnout, zda nějaká již z existujících integrací není vhodná k použití v testovací knihovně.

Testovací knihovna je sepsána v jazyce C#, proto k vyhledání již existujících řešení byl použit software NuGet [31], který slouží jako správce balíčků, tedy externích knihoven, v jazyce C# a ekosystému .NET. Z dostupných knihoven na první pohled byly nejzajímavějšími knihovny Docker.Dotnet [32] a Fluent.Docker [33].

Docker.Dotnet je open-source knihovna vytvořena a spravována .NET Foundation. Tato nezisková organizace, založena společností Microsoft, se stará o zlepšování open-source ekosystému okolo .NET platformy [34]. Lze ji tedy v podstatě označit jako „oficiální“ integraci softwaru Docker do .NET ekosystému, přestože je fakticky komunitním dílem.

Oproti tomu knihovna Fluent.Docker je open-source knihovna původně od vývojáře Mario Toffia, na které se ale k dnešnímu dni podílelo určitým způsobem již 30 vývojářů. Knihovna se primárně zaměřuje na integraci tzv. Fluent rozhraní, které dovoluje řetězit metody za sebou, díky tomu, že každá metoda vrací instanci třídy, ze které je metoda volána. [35]

Obě knihovny jsou ve spoustě aspektech srovnatelné - obě jsou open-source knihovny vytvořeny komunitou a obě jsou vhodné pro komerční použití. K integraci byla ovšem zvolena knihovna Fluent.Docker. Tato knihovna obsahuje větší abstrakci jednotlivých příkazů a je tedy mnohem jednoduší k použití a integraci.

5.2.2 Správa sítě

Správu virtualizované sítě má v nové knihovně na starosti třída `NetworkManager`, jež se stará o vytváření, nastavení a nakonec i destrukci sítě. Samotná realizace vytváření a destrukce sítí v prostředí softwaru Docker je ovšem realizována prostřednictvím třídy `DockerNetworkDriver`. Tato třída obsahuje metodu `CreateNetwork(string, int)`, kde jako argumenty požaduje název sítě a počet dostupných IP adres. Toto číslo ovšem musí zahrnovat i počet implicitně obsazených IP adres - tedy IP adresu sítě, broadcast adresu a pro Docker i IP adresu směrovače. Informace o počtu potřebných implicitních zařízeních je uložena ve statické třídě `NetworkConstants`.

Metoda `CreateNetwork` nejdříve zjistí, jaké rozsahy IP adres jsou obsazené a jaké rozsahy jsou volné. Následně na základě těchto informací vytvoří nejmenší možnou síť, do které se vejde požadovaný počet IP adres. Zároveň se třída snaží co nejvíce vyplňovat mezery mezi adresními prostory. Metoda následně pouze vrací instanci třídy `NetworkInfo`, která obsahuje všechny potřebné informace o dané síti. Třída `DockerNetworkDriver` si udržuje instance všech vytvořených sítí, aby byla schopna následně při ukončování vytvořené sítě smazat.

Třída `NetworkManager` má k vytvoření sítě tyto tři metody:

1. `AddNode(string)` - přidání zařízení (kontejneru) do správce, kde argumentem je identifikátor zařízení
2. `AddConnection(string, string)` - přidání propojení mezi dvěma zařízeními, kde argumentem jsou identifikátory zařízení, mezi kterými má být spojení
3. `BuildNetwork()` - metoda, která spustí výpočet a následné vytvoření sítě ve virtualizovaném prostředí

Pro vytvoření úspěšného propojení mezi dvěma zařízeními je potřeba přidat všechna zařízení za pomoci funkce `AddNode`, jinak třída vyhodí výjimku při sestavování sítě. Po zaregistrovaní všech zařízení a všech propojení je zavolána metoda `BuildNetwork`.

Tato metoda nejdříve vytvoří strukturu uložení všech informací o síti a kontejnerech. Metody `AddNode` a `AddConnection` totiž pouze přidají dané informace do front, které jsou v tento moment zpracovávány za pomoci metody `ConstructDataStructure`. Metoda nejdříve pomocí metody `RegisterNode` zaregistruje všechna zařízení, což vede hlavně k přiřazení unikátního indexu všem zařízením. Reference mezi jménem a indexem je uložena ve slovníku `nodeMapper`.

Následně s pomocí metody `RegisterConnection` je zaregistrováno každé spojení, což znamená, že všechny dvojice indexů zařízení, mezi kterými má existovat spojení, jsou přidány do seřazené množiny, která je uložena v atributu `connections`.

Tyto dva indexy jsou ovšem v relaci a platí $\forall x, y \in K, xSy \Rightarrow ySx$, kde K představuje množinu všech indexů zařízení a S je binární relace spojení zařízení zařízení. Proto jsou do atributu `connections` ukládány obě symetrické hodnoty, primárně pro zjednodušení vyhledávání v této kolekci.

Jednotlivé informace o síťovém nastavení zařízení jsou uloženy v seznamu `nodeConfigurations`, který je indexován za pomoci atributu `nodeMapper`. Ten obsahuje instance třídy `ContainerNetworkConf`, která obsahuje informace o všech sítích, ke kterým je dané zařízení připojeno, a jakou v něm má přiřazenou IP adresu, o adrese implicitního směrovače a seznam všech statických směrování. Statická směrování jsou reprezentována za pomoci třídy `NetworkRoute`, jejíž instance obsahuje všechny potřebné informace.

Samotné reálné vytváření sítě počíná vytvořením tzv. správcovské sítě, do které budou připojena všechna zařízení. Ta slouží primárně pro případnou komunikaci s testovací službou. Tato síť je typu bridge. Všechna zařízení budou mít nastavena směrovač této sítě jako implicitní bránu.

Následně třída pro každé spojení vytvoří separátní síť typu bridge. Na první pohled se může zdát, že se jedná o velice neefektivní řešení. Toto řešení bylo ovšem zvolenu z důvodu technického omezení. V každé síti typu bridge je vytvořen implicitní směrovač, který rozesílá komunikaci daným kontejnerům. Tím pádem jde každá komunikace v základním nastavení z odchozího kontejneru do směrovače a následně do cílového kontejneru. Tím by ale byla porušena požadovaná topologie. Z tohoto důvodu je pro každý spoj vytvořena nová síť. Za pomoci statického směrování je poté následně možné směrovat komunikaci do každého zařízení dle požadované topologie.

Metoda tedy nejdříve vytvoří s pomocí metody `CreateNetwork` pro každé připojení síť a informace o dané síti uloží seznamu `networkInfos`. Index sítě je následně uložen do slovníku `networkIndexer`, kde klíčem je dvojice indexů zařízení, mezi kterými daná síť, a tedy spojení, existuje. Oproti atributu `connections`, slovník neobsahuje obě symetrické hodnoty indexů. Platí, že $\forall x \forall y; x, y \in K; (x, y) \in$ `networkIndexer.Keys` $\Rightarrow x < y$.

Po vytvoření všech sítí metoda vypočítá statického směrování pro každé zařízení. Tento proces si zaslouží přiblížení. Metoda iteruje pro všechny zařízení všechny položky ve slovníku `networkIndexer`. Pokud mezi zařízeními existuje přímé spojení tak směrování je jednoduché, daná síť je směrována na dané přilehlé zařízení. Problem nastává, pokud síť není přímo připojena k zařízení. V tento moment je potřeba zjistit, jakým „směrem“ má být zpráva odeslána pro úspěšné doručení.

Nejbližší cesta do dané sítě je zjišťována za pomoci algoritmu BFS, jenž je implementován ve stejnojmenné metodě. BFS, neboli Breadth first search, funguje na principu prohledávání do šířky. Algoritmus nejdříve obdrží informace o počátečním a konečném zařízení, tedy o uzlech, mezi kterými má nalézt cestu.

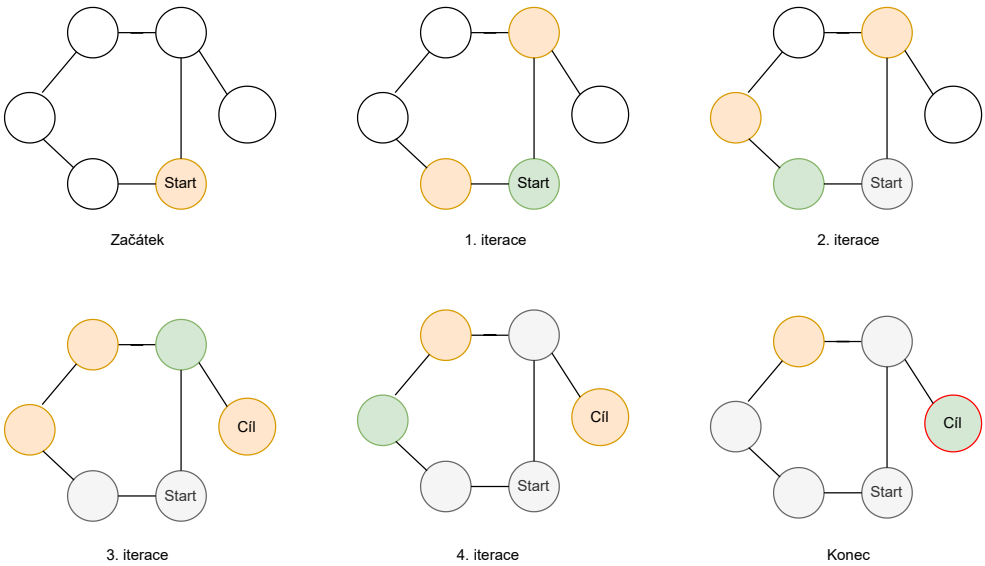
V počátku si algoritmus vytvoří frontu uzlů ke zpracování a seznam navštívených uzlů. Následně je do fronty vložen počáteční uzel a označen jako na-

vštívený. Algoritmus následně ve smyčce skrz všechny položky fronty zkoumá jednotlivé uzly. Po vyjmutí prvního uzlu z fronty algoritmus zkoumá všechny jeho sousední uzly. Pokud sousední uzel ještě nebyl navštíven, tak poté je přidán na konec fronty. Po prozkoumání všech sousedních uzlů algoritmus pokračuje na další položku ve frontě. Součástí jednotlivých položek ve frontě je mimo informace o uzlu, který má být zpracován, také seznam uzlů, který reprezentuje cestu do daného uzlu ke zpracování z počátečního uzlu. V případě, kdy uzel, který má být zpracován, je shodný s koncovým uzlem, je navrácen seznam uzlů reprezentující cestu do koncového uzlu. Pokud není nalezena cesta, poté metoda navrací prázdný seznam. [36]

Ukázkový běh algoritmu můžeme vidět na obrázku 5.1. Na něm lze vidět jednotlivé iterace algoritmu, kde stav každého uzlu je označen těmito barvami:

- Bílá - nezpracován
- Zelená - aktuálně zpracováván
- Oranžová - ve frontě ke zpracování
- Šedá - zpracován

Jak je vidět, algoritmus postupně nachází všechny uzly od počátečního dokud nenalezne cílový uzel.



Obrázek 5.1: Běh algoritmu BFS

Pomocí tohoto algoritmu je tedy zjištěna nejkratší cesta do dané sítě. Tato cesta je nalezena v nejhorším případě v komplexitě $O(|V| + |E|)$, kde $|V|$ je počet uzlů, neboli kontejnerizovaných zařízení, a $|E|$ je počet hran mezi uzly,

neboli spojení mezi zařízeními. [36] Zároveň metoda pro každé zařízení může být volána maximálně $|V| - 1$ krát. Tyto vlastnosti jsou naprosto dostačující pro aktuální použití.

Metoda tedy navrací seznam zařízení, bez počátečního zařízení, který reprezentuje cestu od počátečního zařízení do cílového zařízení, neboli do dané sítě. Podstatnou informací je první položka v seznamu, tedy to sousední zařízení, kterému musí být zpráva odeslaná z daného zařízení, aby dosáhla cílového. To je nastaveno jako směr, kterým mají být všechny zprávy do dané sítě odesílány. Tímto způsobem implementace je testovací knihovna schopna nastavit všechny požadované topologie. Zároveň vždy odesílá zprávy nejkratší možnou cestou.

5.2.3 Správa kontejnerů

Stejně jako síť, i kontejnery mají svého správce, realizovaného třídou `ContainerManager`. Každý kontejner je podobně jako v případě sítí registrován s pomocí metody `AddContainer(string, ContainerConf)`, kde metoda obdrží v argumentech název a nastavení zařízení získaného z konfiguračního souboru. To je uloženo do fronty pro následné zpracování.

Po přidání všech zařízení je možné zavolat metodu `Build`. Ta očekává jako argument instanci třídy `NetworkManager`, která již má vypočítané nastavení sítě daných zařízení. Nastavení pro jednotlivá zařízení jsou získávána za pomoci metody `NetworkManager.GetContainerConf(string)`, kde argumentem je název zařízení.

Metoda `Build` následně zpracovává všechna zařízení ve frontě. Pokud je obraz na zařízení definován za pomoci Dockerfile, tak poté před vytvořením zařízení metoda `BuildImage` sestaví daný obraz. Metoda nejdříve zkontroluje, zdali daný obraz z Dockerfile již nevytvořila. Tuto kontrolu provádí tím, že při vytvoření si uloží hash zdrojového souboru do slovníku `imageTags`, kde hash souboru je klíčem a následně název vytvořeného obrazu je hodnotou ve slovníku. Po zajištění existence obrazu tedy metoda `BuildImage` vrátí název odpovídajícího obrazu.

Vytvoření samotného kontejneru je následně provedeno metodou `CreateContainer`. Ta dostane jako argument název zařízení, název obrazu, instanci nastavení zařízení a instanci nastavení sítě zařízení.

Každý kontejner je definován za pomoci rozhraní `IVMContainer`, definovaného v sekci 4.2.2. Pomocí těchto metod je každý kontejner nastavován a realizace těchto nastavení je poté na dané implementaci rozhraní. V aktuální implementaci existují tyto dvě implementace kontejnerů:

- `DefaultContainer` - základní kontejner vycházející z operačního systému Ubuntu 22.04. Reprezentován je hodnotou z výčtového typu `ContainerType.DEFAULT`

- `NetworkLogger` - kontejner, který zajišťuje odposlouchávání sítě. Reprezentován je hodnotou z výčtového typu `ContainerType.NETWORK_LOGGER`

Oba dva typy kontejnerů jsou rozlišeny za pomoci výčtového typu `ContainerType`. Každá jeho hodnota by měla být napsána velkými písmeny. V konfiguračním souboru se následně preferuje verze s malými písmeny, přestože jakákoliv kombinace velkých a malých písmen bude akceptována.

Následně rozšiřující statická metoda výčtového typu `GetInstance` dokáže vytvořit instanci daného typu kontejneru. Metoda zároveň obdrží argumenty pro konstruktor daného kontejneru. Hlavními argumenty jsou název kontejneru a název obrazu kontejneru. Kontejner typu `NetworkLogger` může ještě obdržet instanci třídy `ServiceConfiguration`, díky které obdrží nastavení testovací služby, ke které se následně připojí.

Metoda `CreateContainer` následně vrátí instanci kontejneru, která je následně uložena do slovníku `containers`, kde klíčem je název zařízení.

5.2.4 Správce celého virtualizovaného prostředí

Předchozí zmíněné správce je potřeba orchestrovat společně. K tomuto slouží třída `VirtualEnvironmentManager`, která je dle návrhu požadovaný správce virtualizovaného prostředí. Třída obdrží instanci `EnvironmentConfiguration`, která reprezentuje nastavení virtualizovaného prostředí definovaného v sekci 4.2.1. Následně také obdrží instanci třídy `ServiceConfiguration`, ve které je obsaženo nastavení testovací služby.

Třída v sobě obsahuje instance obou dvou správců `NetworkManager` a `ContainerManager`. V konstruktoru je zavolána metoda `SetupEnvironment`, která zaregistruje všechny zařízení a připojení mezi nimi. Zároveň, pokud je dle nastavení potřeba zachytávat na daném spojení komunikaci, tak také přidá adekvátně dané odposlouchávače komunikace.

Zavoláním metody `Build` je následně vytvořeno celé prostředí. Nejdříve je vytvořena požadovaná síť a poté jsou vytvořeny všechny kontejnery. Na konci metoda spustí všechny kontejnery.

Za pomoci metody `Stop` je poté možné zastavit všechny kontejnery. Nakonec, metodou `Dispose` jsou následně odstraněny všechny vytvořené prostředky prostřednictvím daných správců.

5.2.5 Ošetření chyb

Implementace virtualizovaného prostředí ošetřuje všechny chyby tím, že v případě jejich vzniku vyhodí odpovídající výjimku. Je tedy potřeba tyto chyby v případě jejich vzniku následně zachytávat a adekvátně vyhodnotit.

Ovšem pokud je program násilně ukončen, může dojít k nesmazání daných vytvořených prostředků v Docker prostředí. To může způsobit jejich akumulaci

a v případě sítě dokonce selhání vytvoření sítě. Testovací knihovna proto definuje ve statické třídě `TestLibConstants` proměnnou `ResourcePrefix`, která je přidána ke všem vytvořeným prostředkům.

Díky tomu lze identifikovat ty prostředky, jež byly vytvořeny pomocí knihovny. Každý správce má tedy implementovanou metodu pro odstranění zbylých prostředků. Ty jsou volány prostřednictvím metody `RemoveHangingResources` v konstruktoru správce virtualizovaného prostředí před započetím všech akcí k vytvoření nového prostředí.

Z tohoto přístupu je ovšem očividné, že testovací knihovna nedokáže rozlišit mezi zdroji, které jsou využívané jinou instancí knihovny, a nesmazanými zdroji. Z tohoto důvodu může na jednom stroji běžet pouze jedna instance knihovny.

5.2.6 Definice kontejnerů

Jak jsem zmínil v sekci 5.2.3, aktuálně testovací knihovna definuje dvě zařízení. Zařízení typu `default` je výchozím zařízením pro všechny aktuální kontejnery. Zařízení vychází z operačního systému Ubuntu 22.04.

Ke správné orchestraci zařízení potřebuje kontejnerizované zařízení obsahovat potřebné nástroje k nastavení sítě. Všechny tyto nástroje jsou definované v novém obraze nazvaném `testlib-ubuntu-base`. Obraz obsahuje tyto balíčky:

- `net-tools`
- `iproute2`

Všechny kontejnery, které budou vycházet z tohoto obrazu, bude možné použít v testovací knihovně. Tento obraz byl zveřejněn ve veřejném repositáři na Docker Hub [19], kde je dostupný pod názvem `wheeeper/testlib-ubuntu-base`.

Označení verze obrazu bude kopírovat označení podkladového obrazu operačního systému Ubuntu. Tedy verze bude stejná jako verze operačního systému Ubuntu.

Následná definice kontejnerů v konfiguračním souboru virtualizovaného prostředí bude probíhat podle definice v 4.2.1. Je ovšem potřeba zajistit, že všechny cesty v konfiguraci budou pro program dostupné, či validní. V případě relativních cest uvnitř konfiguračního souboru bude knihovna pokládat umístění konfiguračního souboru jako výchozí bod pro dané relativní cesty.

5.2.7 Odposlouchávání komunikace

K odposlouchávání komunikace byl vytvořen nový program pod názvem `NetworkLogger`. Program využívá knihovnu SharpPcap [37] pro zachytávání komunikace na daném virtuálním rozhraní.

Program přijímá variabilní počet argumentů v závislosti na módu fungování. Program podporuje dva módy fungování

1. Mód bez připojení k testovací službě
2. Mód s připojením k testovací službě

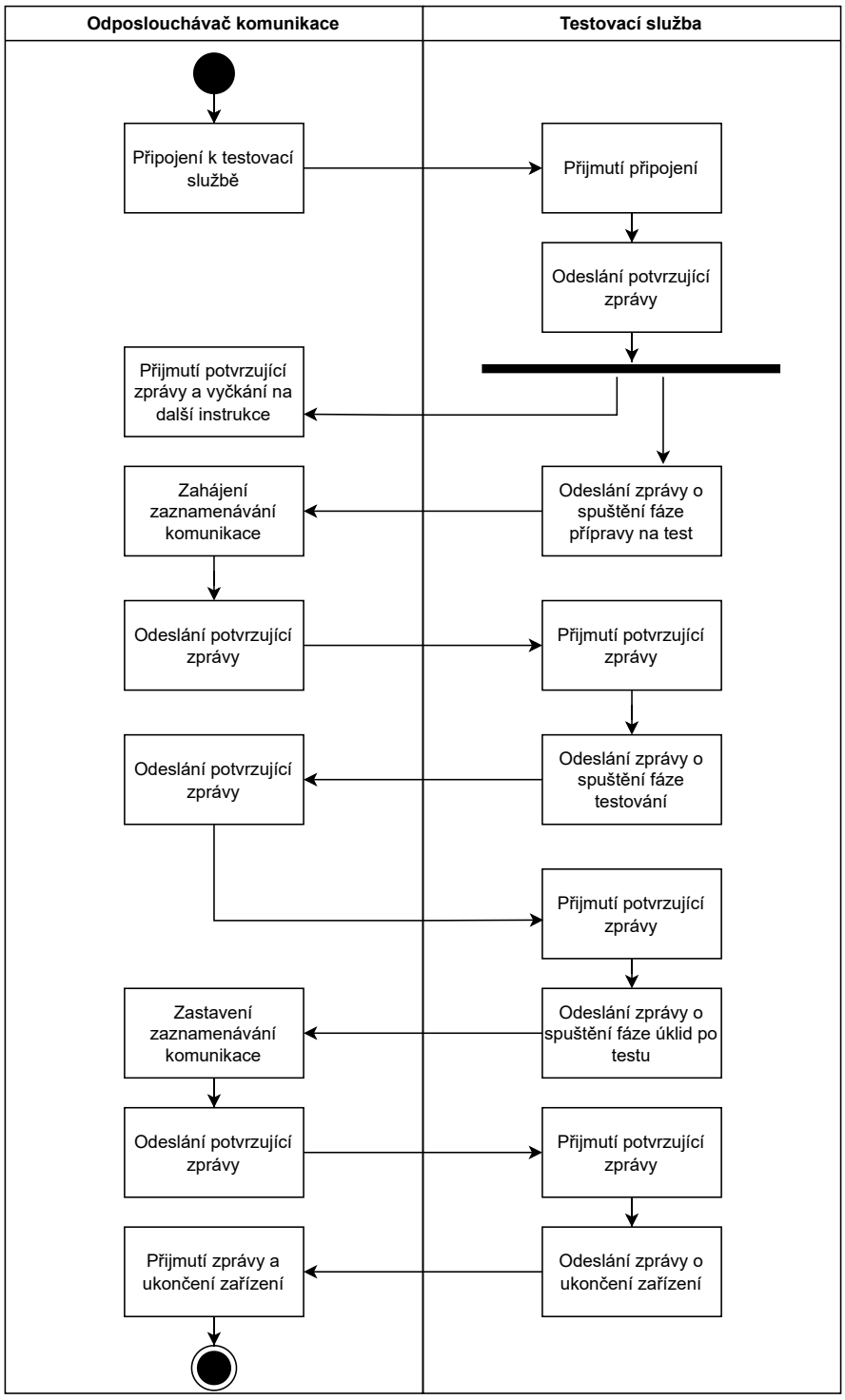
K aktivaci prvního módu je potřeba předat programu přepínač `--no-service`, který mód aktivuje. Následně program očekává seznam názvů rozhraní, oddělených mezerou, ze kterých má zachytávat komunikaci. Při spuštění program zachytává všechnu komunikaci na daných rozhraní a následně vypisuje jednotlivé zachycené zprávy na standardní výstup.

V druhém implicitně definovaném módu testovací knihovna očekává tyto poziční argumenty:

1. Složku, kde bude ukládat zachycené zprávy do souboru.
2. Adresu, na které běží testovací služba.
3. Port, na kterém běží testovací služba.
4. Variabilní počet jmen rozhraní, které má zaznamenávat, oddělený mezerou.

Ilustraci úspěšného běhu s jedním testem můžeme vidět na diagramu aktivit, který je na obrázku 5.2. Jak je vidět, po obdržení argumentů se program připojí k testovací službě a projde stejnou inicializační fází, jako všichni ostatní účastníci testu. Následně po obdržení zprávy o započetí testu začne zaznamenávat síť. Program využívá separátních vláken pro záznam komunikace a pro zpracovaní jednotlivých paketů. Díky tomuto přístupu není omezen výkon zaznamenávání například zápisem do souboru. Všechny přijaté pakety mohou být tedy ihned zapsány.

Zaznamenávání sítě je ukončeno při obdržení zprávy o ukončení testu. Jednotlivé testy jsou tedy zaznamenávány separátně a pro každý test je vytvořen separátní záznam, rozlišen identifikátorem testu. Zároveň je ovšem kontrolována kolize záznamů a jméno výstupního souboru je v případě kolize upraveno.



Obrázek 5.2: Aktivity diagram odposlouchávače komunikace

Program má definovaný Dockerfile, vycházející z předem definovaného obrazu `testlib-ubuntu-base`, na jehož základě může být program spuštěn ve virtualizovaném prostředí prostřednictvím softwaru Docker. Ten je oproti původnímu prostředí rozšířen o dvě komponenty:

- `dotnet-runtime-6.0` - vše potřebné ke spuštění .NET 6 programu v Ubuntu
- `libpcap-dev` - knihovna k zaznamenávání packetů

Obraz také využívá obraz `mcr.microsoft.com/dotnet/sdk:6.0`, s jehož pomocí je celý program zkompilován. V základním nastavení je program spuštěn v módu bez připojení k testovací službě. To lze při vytváření kontejneru změnit a tedy program správně nastavit. Daný vytvořený obraz byl zveřejněn ve veřejném repositáři na Docker Hub [19] a je dostupný pod názvem `wheeeper/networklogger`.

5.3 Integrace virtualizovaného prostředí do testovací knihovny

Pro integraci virtualizovaného prostředí je potřeba integrovat virtualizované prostředí do služeb testovací knihovny. Původní třída `ServiceRunner` byla přejmenována na `ServicesRunner`, aby název lépe reflektoval její zónu odpovědnosti. Ta v sobě obsahuje instance tříd `TestService` a `VirtualEnvironmentManager`.

Do statické třídy `API` byla přidána statická metoda `BuildEnvironment`. Tato metoda obdrží v argumentu cestu ke konfiguračnímu souboru daného virtualizovaného prostředí. Metoda načte danou konfiguraci a následně zavolá metodu `ServiceRunner.StartServices(EnvironmentConfiguration)`. S její pomocí je vytvořeno dané virtualizované prostředí. Nejdříve je ovšem zkontrolováno, zdali dané prostředí již není vytvořeno. Pokud ano, metoda ponechává dříve vytvořené prostředí. V opačném případě metoda ukončí testovací službu a zničí předchozí virtualizované prostředí (pokud nějaké existuje). Následně vytvoří nové virtualizované prostředí. Po úspěšném vytvoření prostředí je poté spuštěna testovací služba.

Použití této funkce je zamýšleno v rámci jednotlivých tříd testů. Testovací knihovna používá knihovnu MSTest pro propojení se serverem Azure DevOps, který může automaticky spouštět jednotlivé testy. Všechny testy musí být shlukovány do jednotlivých testovacích tříd. Testovací knihovna MSTest definuje atribut `ClassInitialize`. Metoda s tímto atributem bude spouštěna právě jednou před započetím testů v dané třídě. Je tedy logické shlukovat jednotlivé testy dle konfigurace virtualizovaného prostředí. Tímto použitím se minimalizuje riziko zbytečného přestavování virtualizovaného prostředí.

Třída `API` také nově definuje metodu `SetServiceConfig`, díky které může být změněno základní nastavení testovací služby, jež je popsáno v sekci 5.1.1.

Kapitola 6

Demonstrace použití knihovny

Tato kapitola popisuje ukázku použití nově vytvořené testovací knihovny.

6.1 Predispozice ke spuštění testovací knihovny

K tomu, aby mohla být testovací knihovna spuštěna, musí mít stroj nainstalované tyto položky:

- .NET 6 včetně správce balíčků NuGet
- Docker Desktop verze 4.17.1 a vyšší

Zároveň pro Docker Daemon, jenž je součástí Docker Desktop, musí být nastaveny rozmezí IP adres, ve kterých lze vytvářet sítě. To lze upravit v Docker Desktop v nastavení pod kolonkou `Docker Engine`. Možnou ukázku dodatečného nastavení můžeme vidět na výpisu 6.1. Současně se všemi těmito požadavky musí být zaručeno, že v průběhu běhu knihovny nebude spuštěna další instance testovací knihovny.

```
"default-address-pools": [
    {
      "base": "172.17.0.0/12",
      "size": 20
    },
    {
      "base": "192.168.0.0/16",
      "size": 24
    }
]
```

Výpis 6.1: Nastavení rozmezí IP adres pro Docker

6.2 Definice testu

Hlavní výhodou nové testovací knihovny je její flexibilita ve vytváření jednotlivých topologií ve virtualizovaném prostředí. Cílem tedy bylo navrhnout test, který by byl lehce škálovatelný a použitelný na všechny topologie.

Pro komunikaci mezi zařízeními byl zvolen průmyslový protokol ModbusTCP [38]. Všechna komunikace mezi zařízeními bude probíhat dle jeho specifikace. K úspěšné komunikaci tedy bude potřeba právě jeden server a alespoň jeden klient, který bude se serverem komunikovat. Test poté bude probíhat následovně:

1. Uvnitř testovacího prostředí bude běžet server, který bude přijímat komunikaci.
2. Klient přečte hodnotu registru, který je nazýván jako tzv. *coil*, je na adrese 0 a má velikost právě jeden bit. Pokud je jeho hodnota rovna 0, poté se pokusí změnit jeho hodnotu na hodnotu 1. V případě neúspěchu je tento krok opakován.
3. Klient přečte první čtyři tzv. *holding* registry, které mají adresu 0-3.
4. Klient inkrementuje získanou hodnotu registrů. Pokud hodnota inkrementací překročí maximální povolenou hodnotu v registrech, je registr nastaven na hodnotu 0.
5. Klient zapíše nové hodnoty zpátky do daných registrů.
6. Klient znovu přečte hodnotu z daných registrů a zkontroluje, zda byly nové hodnoty správně zapsány.
7. Klient nastaví registr, který nastavil v druhém kroku zpátky na hodnotu 0.

Tento test je jednoduše škálovatelný na libovolný počet klientů. Každý klient bude provádět kroky 2 až 7. Je ale ovšem potřeba zajistit exkluzivní přístup k daným registrům, tedy pouze jeden klient může v danou chvíli manipulovat s danými holding registry.

K tomu slouží krok 2 a 7, kdy coil registr na adrese 0 slouží jako zámek. Pokud je tento registr úspěšně nastaven na hodnotu 1, tak poté pouze tento klient, jenž tuto hodnotu nastavil, může zapisovat do daných registrů. V opačném případě bude požadavek na zápis odmítnut.

6.3 Implementace testu

Implementace testu vyžaduje implementaci dvou zařízení - serveru a klienta. Jazykem implementace pro obě zařízení byl zvolen jazyk Python. Obě zařízení budou pro komunikaci využívat knihovnu pyModbusTCP [39]. Tato

knihovna obsahuje implementaci námi požadovaného průmyslového protokolu ModbusTCP, a to jak pro server, tak pro klienta.

6.3.1 Server

Na straně serveru je potřeba zajistit exkluzivní přístup pro jednotlivé klienty, tedy aby pouze jeden klient mohl manipulovat s registry. Knihovna pyModbusTCP definuje tyto tři třídy:

- `ModbusServer` - logika serveru
- `DataBank` - třída, která uchovává data registrů a zajišťuje bezpečný přístup k nim
- `DataHandler` - třída, která mapuje volání serveru na na metody třídy DataBank

Knihovna přímo podporuje použití vlastní implementace tříd `DataBank` a `DataHandler`. Třída `DataHandler` obsahuje pro zápis tyto dvě funkce:

- `write_coils` - zápis do coil registrů
- `write_h_regs` - zápis do holding registrů

Do obou těchto funkcí byla přidána kontrola, zda má daný klient povolení zapisovat. To je kontrolováno za pomoci IP adresy. Funkce `write_coils` ovšem dovoluje zapisovat i v případě, že registr na adrese 0 má hodnotu 0. Pokud je hodnota tohoto registru nastavena na 1, je poté uložena IP adresa zařízení, které danou změnu provedlo. Symetricky, pokud je hodnota téhož registru nastavena na hodnotu nula, tak poté je IP adresa smazána a je povolen zápis do coil registru dalšímu klientovi. Funkce zároveň obsahuje zámek pro zajištění exkluzivního přístupu při zamykání a odemykání.

Logika třídy `DataBank` byla nepozměněna. Jedinou změnou byla implementace metod `on_coils_change` a `on_holding_registers_change`, za jejichž pomoci je vypsána každá změna v příslušných registrech.

6.3.2 Klient

Pro orchestraci musí být klient připojen k testovací službě. K tomu bude využita python implementace testovací knihovny, která obsahuje vše potřebné k vytvoření spojení a řízení zařízení na základě instrukcí od testovací služby.

K umožnění testování je potřeba provést tyto věci:

1. Definovat výčtový typ, jenž bude definovat test. Jeho číselná hodnota musí být stejná jak na zařízení, tak v testovacím projektu, kde poběží testovací knihovna.

2. Implementovat testovací případ, který bude vycházet z abstraktní třídy `TestCase`.

3. Vytvořit funkci, která po obdržení identifikátoru testu v argumentu funkce navrátí instanci testu, jenž daný identifikátor reprezentuje.

V implementaci klienta výčtovým typem, který definuje testy, je třída `TestList`. Následně třída `TestModbusRead` definuje daný testovací případ. V neposlední řadě funkce `get_test(int)` převádí identifikátor na instanci testovacího případu.

Do souboru `main.py`, který je hlavním zdrojovým souborem programu, bylo také přidáno získání přepínačů programu, jež jsou získány prostřednictvím knihovny `argparse`. Program má tyto dva přepínače:

- `-s`, `--service` - IP adresa nebo doména testovací služby
- `-p`, `--port` - port testovací služby

Program díky těmto přepínačům obdrží informace o tom, kde běží testovací služba, na kterou se při jejím spuštění připojí. Spuštění `TestRunner`, třídy, která se stará o běh zařízení, pak můžeme vidět na výpisu 6.2. Jako první je běh inicializován za pomoci funkce `init`, díky čemuž proběhne inicializační fáze připojení se k testovací službě.

Následně je zavolána funkce `handle_instructions`, která čeká na instrukce od testovací služby a ty následně obsluhuje. Běh zařízení je ukončen po obdržení adekvátní zprávy o ukončení testování.

```
args = args.parse_args()
testrunner = TestRunner(get_test)
testrunner.init(args.service, args.port)
testrunner.handle_instructions()
```

Výpis 6.2: Spuštění řízení testovaného zařízení

6.3.3 Definice docker kontejnerů

Díky podobné struktuře obou programů jsou všechny kontejnery skoro identické, až na možnou změnu v kopírování zdrojových souborů. Kontejnery vychází z předem definovaného obrazu `testlib-ubuntu-base`. Tento obraz je následně rozšířen o tyto balíčky:

- `python-3.10`
- `python-3.10-venv`

Tyto balíčky jsou nutnou součástí kontejneru a slouží ke spuštění dříve definovaných Python programů. Následně je vytvořena složka `app`, do které jsou nakopírovány všechny zdrojové soubory programu. Spolu s ní je vytvořena složka `packages`, do které jsou nakopírovány všechny balíčky ze stejnojmenné složky v kořenové složce programu. Tato složka slouží pro balíčky, které nebyly uveřejněny a tedy nejsou dostupné z veřejného repositáře.

Po nakopírování všech souborů je vytvořeno virtuální Python prostředí. S jeho pomocí jsou poté nainstalovány všechny lokální balíčky ve složce packages, pokud nějaké existují. Poté jsou nainstalovány všechny balíčky obsažené v souboru `requirements.txt`. Ten obsahuje seznam všech potřebných balíčků ke spuštění programu, včetně těch, jež nejsou veřejně dostupné. Pokud je možné program spustit bez argumentů, tak je do kontejneru přidán i příkaz na spuštění aplikace. Při použití v knihovně ovšem bude toto spuštění změněno. Ukázku definice docker kontejneru můžeme vidět na výpise 6.3

```
FROM wheeeper/testlib-ubuntu-base:22.04 AS base
RUN apt-get update && apt-get upgrade -y
RUN apt-get update && apt-get install -y python3.10 python3.10-venv
WORKDIR /app
COPY "src/" "/app/src"
COPY ["main.py", "requirements.txt", "/app/"]

RUN mkdir /packages
COPY "./packages/" "/packages/"
RUN python3.10 -m venv /opt/venv

RUN /opt/venv/bin/pip install  /packages/** || true
RUN /opt/venv/bin/pip install -r /app/requirements.txt
CMD ["/opt/venv/bin/python", "/app/main.py"]
```

Výpis 6.3: Ukázka definice kontejneru

6.3.4 Definice virtualizovaného prostředí

Ke spuštění zařízení ve virtualizovaném prostředí je potřeba definovat nastavení virtualizovaného prostředí, jež bylo popsáno v sekci 4.2.1. K testování na všech třech požadovaných topologiích budou potřeba minimálně tři konfigurace, které ovšem budou v mnoha částech podobné. První položkou v konfiguraci jsou označení virtualizovaného prostředí. Ty dle topologie budou:

- `modbus_line` - konfigurace s topologií sériové linky
- `modbus_star` - konfigurace s topologií hvězdy
- `modbus_circle` - konfigurace s topologií kruhu

Poté následuje konfigurace počtu očekávaných zařízení. Každá konfigurace bude obsahovat 1 server a 6 klientů. K testovací službě budou připojeni

pouze klienti, tedy testovací služba bude očekávat připojení 6 účastníků testu. Ukázku konfigurace obou položek můžeme vidět na výpisu 6.4

Další položkou v konfiguraci jsou jednotlivá zařízení. Definici jednotlivých zařízení můžeme vidět na výpisu 6.4. Virtualizované prostředí obsahuje dle definice dva typy zařízení - server a klienty. Server je definován pod názvem `modbus_server`, je typu `default` a je definován v `..\server\Dockerfile`. V kolonce `commands` pak můžeme vidět příkaz na spuštění zařízení. Server má také nastavenu kolonku `logging` na hodnotu `true`, čímž bude zaručeno zaznamenávání komunikace mezi serverem a klienty.

Následně lze vidět definice klientů. Ti jsou pojmenování jakožto `modbus_client_[1-6]`. Každá definice klienta je totožná a liší se pouze v názvu zařízení. Klient je definován v `..\client\Dockerfile` a je taktéž typu default. V kolonce `commands` pak lze vidět příkaz na spuštění zařízení, kde v přepínačích jsou předány parametry pro připojení se k testovací službě.

```
label: modbus_(line|star|circle)
service:
  connections: 6
containers:
    modbus_server:
      build:
          dockerfile: ..\server\Dockerfile
      type: default
      commands:
        - "/opt/venv/bin/python /app/main.py"
      logging: true
    modbus_client_1:
      build:
          dockerfile: ..\client\Dockerfile
      type: default
      commands:
        - "/opt/venv/bin/python /app/main.py -s host.docker.internal -p 1337"
        .
        .
        .
    modbus_client_6:
      build:
          dockerfile: ..\client\Dockerfile
      type: default
      commands:
        - "/opt/venv/bin/python /app/main.py -s host.docker.internal -p 1337"
```

Výpis 6.4: Nastavení zařízení v konfiguraci virtualizovaného prostředí

V neposlední řadě je potřeba nadefinovat kolonku `links`, která definuje propojení mezi zařízeními. Definice pro všechny topologie můžeme vidět na výpisu 6.5. Jak je vidět, definice je jednoduchá, intuitivní a každá položka v seznamu definuje právě jedno spojení.

```
# Definice topologie seriové linky
links:
- "modbus_server:modbus_client_1"
- "modbus_client_1:modbus_client_2"
- "modbus_client_2:modbus_client_3"
- "modbus_client_3:modbus_client_4"
- "modbus_client_4:modbus_client_5"
- "modbus_client_5:modbus_client_6"

# Definice topologie hvězdy
links:
- "modbus_server:modbus_client_1"
- "modbus_server:modbus_client_2"
- "modbus_server:modbus_client_3"
- "modbus_server:modbus_client_4"
- "modbus_server:modbus_client_5"
- "modbus_server:modbus_client_6"

# Definice topologie kruhu
links:
- "modbus_server:modbus_client_1"
- "modbus_client_1:modbus_client_2"
- "modbus_client_2:modbus_client_3"
- "modbus_client_3:modbus_client_4"
- "modbus_client_4:modbus_client_5"
- "modbus_client_5:modbus_client_6"
- "modbus_client_6:modbus_server"
```

Výpis 6.5: Nastavení propojení zařízení pro všechny topologie

6.4 Testovací projekt

K tomu, aby mohla být spuštěna testovací knihovna, je zapotřebí vytvořit testovací projekt. Tento projekt musí obsahovat vytvořenou testovací knihovnu a také testovací knihovnu MSTest. Projekt je možné vytvořit ze šablon, jenž jsou poskytnuty IDE Visual Studio 2022 [40], jenž je využito při vývoji všech .NET komponent. K vytvoření projektu byla použita šablona s názvem *MSTest Test Project*. Do vytvořeného projektu je následně přidána testovací knihovna prostřednictvím správce balíčků NuGet [31].

Aby mohl program využívat relativních cest pro všechny cesty konfigurační soubory, je potřeba nakopírovat všechny potřebné soubory do výstupní složky kompilace. K tomu byla do projektového souboru přidána direktiva na zkopírování složky `virtual_env`, která obsahuje všechny potřebné zdroje. Tuto direktivu můžeme vidět na výpisu 6.6

Po přidání testovací knihovny je potřeba zajistit její inicializaci a také její správné ukončení. Tuto definici lze vidět na výpisu 6.7. Tento proces zůstal nepozměněn od původní testovací knihovny. Jak je vidět, pro obě operace existují separátní funkce, které obsahují adekvátní atributy `AssemblyInitialize`, respektive `AssemblyCleanup`. Následně testovací knihovna MSTest zajistí je-

jich spuštění před započetím testování a po ukončení testovaní.

```
<ItemGroup>
    <Content Include="virtual_env\**">
        <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </Content>
</ItemGroup>
```

Výpis 6.6: Direktiva pro zkopírovaní složky při kompilaci

```
[TestClass]
public static class TestLibInit
{
    [AssemblyInitialize()]
    public static void AssemblyInit(TestContext context)
    {
        TestLib.API.AssemblyInit(Assembly.GetExecutingAssembly());
    }

    [AssemblyCleanup()]
    public static void AssemblyCleanup()
    {
        TestLib.API.AssemblyCleanup();
    }
}
```

Výpis 6.7: Inicializace a ukončení testovací knihovny

Následně je zapotřebí definovat výčtový typ, který bude reprezentovat jednotlivé testy. Testovací knihovna požaduje, aby každý výčtový typ obsahoval atribut `TestEnum`. Díky němu může všechny výčtové typy identifikovat a následně kontrolovat kolize mezi jejich číselnými hodnotami. Definici výčtového typy můžeme vidět na výpisu 6.8.

```
[TestEnum]
public enum TestCaseE
{
    TEST_SUCCESS = 0,
    MODBUS_READ = 10,
    TEST_IPERF = 6969
};
```

Výpis 6.8: Definice výčtového typu který identifikuje jednotlivé testy

Jak je vidět, výčtový typ obsahuje tři testy:

- `TEST_SUCCESS` - úspěšný test, který ve všech fázích navrátí hodnotu `true`
- `MODBUS_READUPDATE` - test definovaný v sekci 6.2
- `TEST_IPERF` - test na měření výkonu, jenž je popsán v sekci 7.3

Nyní zbývá jednotlivé třídy dle konvencí testovací knihovny MSTest. Pro každou topologii vytvoříme testovací třídu. Jednotlivé testy v daných třídách budou probíhat na v jednotném virtualizovaném prostředí. Ukázku definice pro topologii sériové linky můžeme vidět na výpisu 6.9. Jako první je definována metoda `StartUp`. Tato metoda je opatřena atributem `ClassInitialize`, který zaručí její zavolání před spuštěním všech testů ve třídě. Následně za pomoci metody `API.BuildEnvironment` je inicializováno dané virtualizované prostředí, kdy v argumentu metoda obdrží cestu ke konfiguračnímu souboru.

Následně je zde metoda `TestReadAndUpdate`, která definuje námi požadovaný test. Tato metoda je opatřena atributem `TestMethod`. Uvnitř metody je následně zavolána metoda `API.Run`, kdy v argumentu je předán identifikátor testu.

```
[TestClass]
public class ModbusLine
{
    [ClassInitialize]
    public static void StartUp(TestContext ctx)
    {
       API.BuildEnvironment(@"virtual_env\modbus\topologies\modbus-line.yaml");
    }

    [TestMethod]
    public void TestReadAndUpdate()
    {
       API.Run(TestCaseE.MODBUS_READUPDATE);
    }
}
```

Výpis 6.9: Ukázka definice testovací třídy

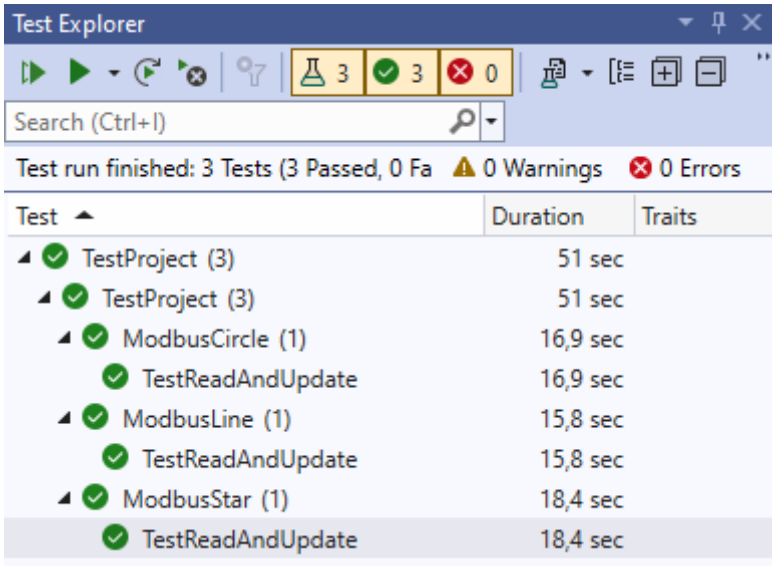
6.5 Spuštění testů

Jednotlivé testy lze spustit přímo jak z příkazové řádky, tak z IDE Visual Studio 2022. Potřebné příkazy pro spuštění skrz příkazovou řádku lze vidět na výpisu 6.10. Jak lze vidět, jako první jsou získány všechny balíčky skrz správce balíčků NuGet. Následně je spuštěno testování, což zahrnuje i kompilaci projektu.

```
nuget restore -ConfigFile nuget.config .\TestProject.csproj
dotnet test
```

Výpis 6.10: Příkazy ke spuštění testů skrz příkazovou řádku

Při využití IDE Visual Studio 2022 lze po zobrazení okna *Test Explorer* vidět všechny testy dostupné v projektu. Ty lze následně ze stejného pohledu i spustit. Po dokončení je následně vidět, zda test proběhl úspěšně nebo ne. Pohled po úspěšném dokončení všech testů můžeme vidět na obrázku 6.1.



Obrázek 6.1: Pohled při úspěšném dokončení všech testů

Po každém testovacím běhu lze v kořenové složce programu nalézt složku se záznamem běhu všech testů. Uvnitř této složky jsou složky dle názvů všech virtualizovaných zařízení, které byly během testování použity. Tyto složky mají název dle kolonky `label` v konfiguraci.

Uvnitř každé složky jsou poté složky s názvy jednotlivých zařízení. Složka je vytvořena právě tehdy, pokud je co ze zařízení uložit, jako například záznam výpisu ze zařízení. V jednotlivých složkách odposlouchávačů komunikace poté nalezneme i záznam komunikace v průběhu testu.

Výstup ze serveru můžeme vidět na výpisu 6.11. Pro kompaktnost z něj byla odstraněna informace o datu a času zápisu dané zprávy. Jak je vidět, všechna zařízení úspěšně provedla změnu v registrech. Zároveň každá změna byla prováděna vždy pouze jedním zařízením zároveň.

Jak již bylo ukázáno u předchozí testovací knihovny [25], takto nastavený projekt lze poté spustit prostřednictvím Azure DevOps a tím zcela automatizovat testovaní. Jednotlivé testy lze přiřadit k testovacím příkazům definovaným v Azure DevOps, a ty se poté dají dle libosti spouštět. Je ovšem samozřejmé, že daný stroj, na kterém testování poběží, musí splňovat podmínky definované v sekci 6.1.

```
Modbus server started
change in coil space [False > True ] at @ 0x0000 from ip: 172.16.16.82
change in hreg space [14671 > 14672] at @ 0x0000 from ip: 172.16.16.82
change in hreg space [36946 > 36947] at @ 0x0001 from ip: 172.16.16.82
change in hreg space [20782 > 20783] at @ 0x0002 from ip: 172.16.16.82
change in hreg space [56603 > 56604] at @ 0x0003 from ip: 172.16.16.82
change in coil space [True  > False] at @ 0x0000 from ip: 172.16.16.82
change in coil space [False > True ] at @ 0x0000 from ip: 172.16.16.51
change in hreg space [14672 > 14673] at @ 0x0000 from ip: 172.16.16.51
change in hreg space [36947 > 36948] at @ 0x0001 from ip: 172.16.16.51
change in hreg space [20783 > 20784] at @ 0x0002 from ip: 172.16.16.51
change in hreg space [56604 > 56605] at @ 0x0003 from ip: 172.16.16.51
change in coil space [True  > False] at @ 0x0000 from ip: 172.16.16.51
change in coil space [False > True ] at @ 0x0000 from ip: 172.16.16.35
change in hreg space [14673 > 14674] at @ 0x0000 from ip: 172.16.16.35
change in hreg space [36948 > 36949] at @ 0x0001 from ip: 172.16.16.35
change in hreg space [20784 > 20785] at @ 0x0002 from ip: 172.16.16.35
change in hreg space [56605 > 56606] at @ 0x0003 from ip: 172.16.16.35
change in coil space [True  > False] at @ 0x0000 from ip: 172.16.16.35
change in coil space [False > True ] at @ 0x0000 from ip: 172.16.16.66
change in hreg space [14674 > 14675] at @ 0x0000 from ip: 172.16.16.66
change in hreg space [36949 > 36950] at @ 0x0001 from ip: 172.16.16.66
change in hreg space [20785 > 20786] at @ 0x0002 from ip: 172.16.16.66
change in hreg space [56606 > 56607] at @ 0x0003 from ip: 172.16.16.66
change in coil space [True  > False] at @ 0x0000 from ip: 172.16.16.66
change in coil space [False > True ] at @ 0x0000 from ip: 172.16.16.74
change in hreg space [14675 > 14676] at @ 0x0000 from ip: 172.16.16.74
change in hreg space [36950 > 36951] at @ 0x0001 from ip: 172.16.16.74
change in hreg space [20786 > 20787] at @ 0x0002 from ip: 172.16.16.74
change in hreg space [56607 > 56608] at @ 0x0003 from ip: 172.16.16.74
change in coil space [True  > False] at @ 0x0000 from ip: 172.16.16.74
change in coil space [False > True ] at @ 0x0000 from ip: 172.16.16.42
change in hreg space [14676 > 14677] at @ 0x0000 from ip: 172.16.16.42
change in hreg space [36951 > 36952] at @ 0x0001 from ip: 172.16.16.42
change in hreg space [20787 > 20788] at @ 0x0002 from ip: 172.16.16.42
change in hreg space [56608 > 56609] at @ 0x0003 from ip: 172.16.16.42
change in coil space [True  > False] at @ 0x0000 from ip: 172.16.16.42
```

Výpis 6.11: Výstup ze serveru po testu

Kapitola 7

Zhodnocení testovací knihovny

Vytvořená testovací knihovna splňuje požadavky a cíle, které na ní byly kladeny, a které byly stanoveny v kapitole 1. Vyvstává zde ovšem otázka, jak efektivní je toto řešení. V následujících sekcích je tedy vytvořené řešení zhodnoceno na základě několika kritérií.

7.1 Kategorie hodnocení

Vytvořená testovací knihovna bude zhodnocena na základě těchto kategorií:

- Flexibilita a možnosti použití knihovny
- Výkon virtualizované sítě
- Rychlost a efektivnost vytváření virtualizovaného prostředí

Všechny následující testy budou prováděny na počítači s těmito parametry:

- Procesor - Intel Core i7-8850H
- Grafická karta - Nvidia Quadro P2000
- RAM - 16 GB
- Operační systém - Windows 10 verze 21H2

7.2 Zhodnocení flexibility a možnosti použití knihovny

Nová testovací knihovna může být využita pro jakékoliv testy, které vyžadují propojení zařízení, neboli kontejnerů, mezi sebou. Díky její implementaci je

možné testy implementovat s libovolným počtem zařízení v různých topologiích.

Knihovna v aktuální podobě podporuje zařízení, které vychází z operačního systému Ubuntu. Tato podpora může být širší, protože závislost na systému Ubuntu je pouze kvůli nástrojům, které umožňují nastavení sítě na zařízení. Seznam kompatibilních systému ovšem zkoumám nebyl. Je zřejmé, že toto snižuje flexibilitu knihovny. Zároveň ovšem není náročné přidat další implementace pro jiné operační systémy.

Prostor ke zlepšení je i v registraci testů. Jednotlivé testy jsou registrovány za pomoci výčtových typů, kde každý test je poté identifikován číselnou hodnotou. Ideálním cílem je, aby všechny testové identifikátory byly definovány na jednom místě. V předchozí implementaci, jež podporovala jazyky C# a C++ byl řešením tohoto problému jeden společný zdrojový soubor, díky čemuž byl problém vyřešen. S příchodem podpory jazyku Python to ale aktuálně není možné. Do budoucna je tedy vhodné investigovat způsob, kterým by byl tento problém vyřešen.

Největším omezením knihovny je omezení, kdy pouze jedna instance testovací knihovny může souběžně běžet na jednom stroji. Na první pohled se může zdát toto až moc omezující požadavek, kterému se dalo předejít za pomoci lepšího návrhu. Ve skutečnosti ale toto omezení i potřebujeme.

Při spuštění jednotlivých testů je potřeba, aby každé zařízení běželo tak jak má a nebylo zpomalováno díky nedostatečnému výkonu zařízení, na kterém testování běží. Samotná knihovna sice nic o požadovaném výkonu neví, ovšem tester dokáže provést odhad, zda jím vytvořené virtuální prostředí bude schopné běžet na designovaném stroji. Po odstranění tohoto omezení by ovšem tento odhad již nebyl možný.

7.3 Test výkonu virtualizované sítě

Před započetím měření výkonu ve virtualizované síti vyvstává jednoduchá otázka - je možné navrhnout takový test, který s použitím testovací knihovny změří výkon virtualizované sítě? Odpověď na tuto otázku je pozitivní.

Na měření výkonu sítě existuje jednoduchý nástroj iPerf3. Tento nástroj podporuje několik protokolů jako TCP nebo UDP a dovoluje nastavit různé parametry komunikace. Tento nástroj funguje na architektuře klient-server a je určen pro použití přes příkazovou řádku.

Zároveň ale existuje Python knihovna iPerf3, která zapouzdřuje tento nástroj pro použití v prostředí Python. To znamená, že ve spojení s Python implementací testovací knihovny jsme schopni vytvořit zařízení, které bude možné orchestrovat za pomoci testovací knihovny.

7.3.1 Účastníci testu

Součástí testu měření výkonu budou tito tři participanti:

- Server
- Klient
- Zařízení pro vyplnění místa

Implementace těchto zařízení je velice podobná implementaci zařízení pro test s protokolem ModbusTCP v sekci 6.3. Návrh serveru a klient jsou totožné, až na záměnu ModbusTCP serveru a klient na server a klient nástroje iPerf3. Pro nastavení parametrů testu je využito základních hodnot, které jsou nastaveny v nástroji iPerf. Jediná změněná hodnota byla délka testu, která byla nastavena na 60 sekund. Jednotlivé definice kontejnerů byly rozšířeny o balíček `libiperf-dev`, který obsahuje implementaci nástroje iPerf3.

Nově ovšem přibylo zařízení pro vyplnění místa. Toto zařízení je aktivně připojené k testovací službě, jeho úkolem ovšem není provádět žádné testování. Zařízení na všechny požadavky o testování odpovídá kladně, i když sám žádné akce neprovádí. Jeho cílem je být výplní mezi serverem a klientem pro simulaci topologií a tím vzdálenosti mezi nimi. K tomuto úkolu by bylo možné použít jakékoliv zařízení, ovšem díky tomu, že zařízení bude aktivně připojeno k testovací službě, se významně zrychlí orchestrace virtualizovaného prostředí a tím celého testování.

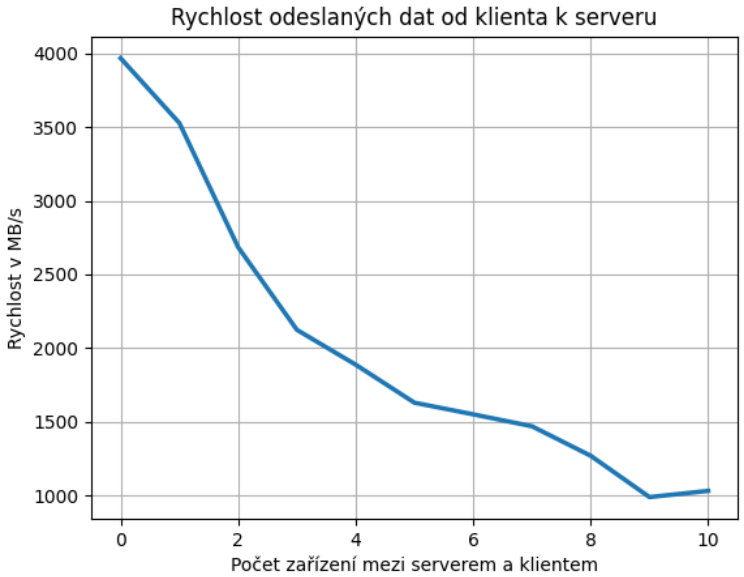
7.3.2 Testované topologie

K testování výkonu sítě byla zvolena topologie sériové linky. Primárním důvodem je, že všechny ostatní sítě se dají při komunikaci mezi serverem a klientem převést na sériovou linku. V případě kruhu zařízení komunikuje jednou cestou, dle nastavení statického směrování sítě. Následně v případě hvězdy je nejvzdálenější zařízení ve vzdálenosti maximálně dva.

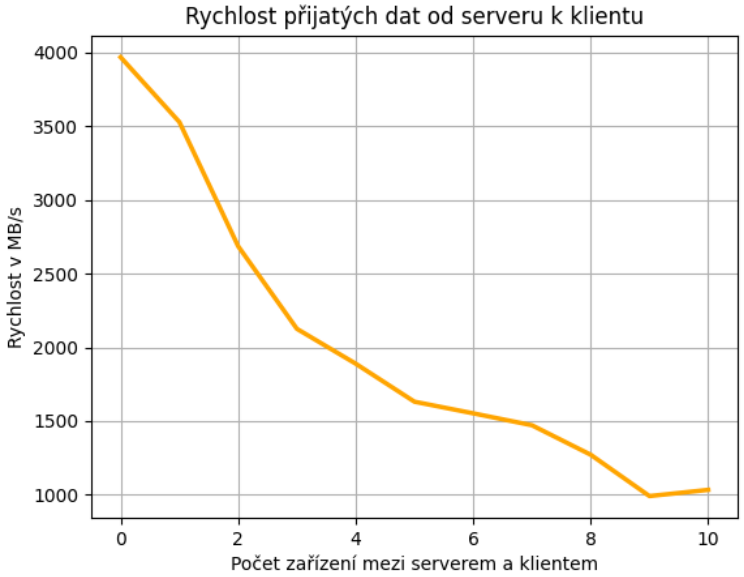
V následujících testech bude tedy 11 topologií, které budou pod označením `line-`x`.yaml`, kde číslo $x \in \{0, 1, \ldots, 10\}$ a značí počet zařízení vyplňujících zařízení mezi serverem a klientem.

7.3.3 Výsledek testu a měření

Výsledky z testů lze vidět na grafech níže. Jak lze vidět na grafu 7.1, respektive 7.2, které znázorňují dosaženou rychlost přenosu dat v MB/s, tak při zvětšující se vzdáleností mezi klientem a serverem se snižuje rychlost přenosu. Nejnižší dosaženou rychlostí byla rychlost 990 MB/s. Tato rychlost je naprosto dostačující pro použití při testování. Na předem zmíněných grafech také můžeme vidět, že rychlost je symetrická. Dané procento počtu opakování přenosu se pohybuje mezi hodnotou 0.5 % a 2.5 %, což je akceptovatelné rozmezí pro využití knihovny.



Obrázek 7.1: Graf rychlosti odeslaných dat od klienta k serveru



Obrázek 7.2: Graf rychlosti přijatých dat od serveru k klientu

Obrázek 7.3: Graf procenta nutných opakování přenosu

7.4 Test rychlosti a efektivnosti vytváření virtualizovaného prostředí

K změření rychlosti vytváření virtualizované sítě byl využit stejný test, jenž byl definován v předchozí sekci. Při orchestraci tohoto testu byla změřena doba potřebná k vytvoření virtualizovaného prostředí a doba potřebná k destrukci vytvořeného virtualizovaného prostředí.

Naměřenou dobu potřebnou k vytvoření virtualizovaného prostředí můžeme vidět na grafu 7.4. Měření času začíná před vstupem do funkce, která spouští vytvoření virtualizovaného prostředí a je ukončeno poté co jsou všechna zařízení spuštěna. Jak lze z grafu vidět, čas potřebný k vytvoření virtualizovaného prostředí roste lineárně.

Je zde ale dobré přiblížit, co se vlastně v tomto celém procesu děje. Tento test byl při vytváření prostředí schopen využít optimalizace, díky které je zabráněno opakovanému sestavování jednoho obrazu. Při každém vytváření virtualizovaného prostředí bylo nutné sestavit maximálně tři obrazy. Velikou část zabere vytváření sítě, respektive získávání informací o aktuálním stavu sítí ve virtualizovaném prostředí. Toto bylo vytvořeno s původní ideou, kdy by na jednom stroji mohlo běžet více instancí knihovny. Do budoucna by tedy bylo vhodné stav sítě ukládat a dotaz na stav sítě provádět pouze v případě chyby, kdy daný rozsah by byl zabrán. Zároveň zde může pomoci i optimalizace, kdy seznam již sestavených obrazů bude ukládán po celou dobu běhu testovací knihovny.

Čas potřebný k destrukci vytvořeného virtualizovaného prostředí lze vidět na obrázku 7.5. Do tohoto času je zahrnuto odeslání zprávy od testovací služby o ukončení testování, které následně vede k ukončení daných zařízení, ukončení testovací služby a následně odstranění vytvořeného virtualizovaného prostředí. Z naměřených dat nelze vyvodit žádnou závislost na počtu vytvořených zařízení. Lze ovšem říci, že destrukce vytvořeného virtualizovaného prostředí v průměru trvala 19,35 sekund.

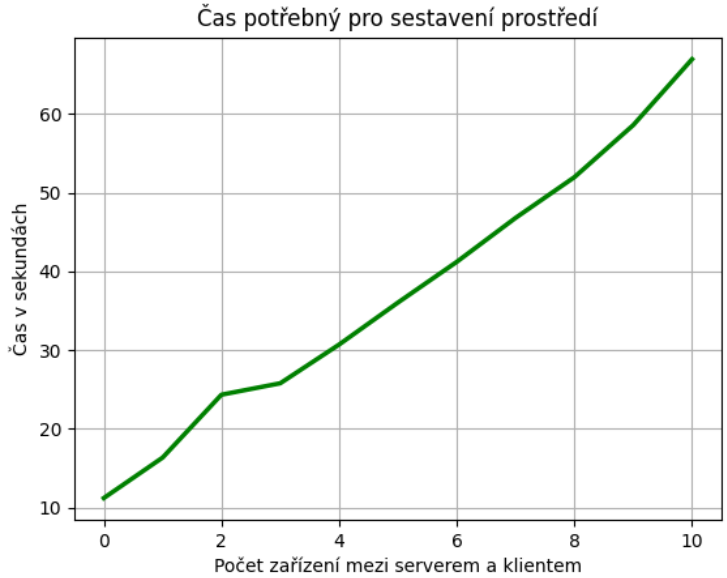
Je ovšem vhodné podotknout, že čím více zařízení není ukončeno s pomocí testovací služby, tím víc narůstá čas potřebný k odstranění vytvořeného virtualizovaného prostředí. Tento nárůst vzniká primárně z časového limitu na nenásilné ukončení, na které software Docker čeká u každého kontejneru. Do budoucna je tedy vhodné investigovat, jak odstranit tento časový limit, jelikož při použití GUI aplikace tento časový limit není přítomný. Při tomto testu bylo potřeba násilně ukončit pouze jedno zařízení. Nutné je ovšem říci, že na funkčnost knihovny toto nemá žádný vliv.

7.5 Shrnutí

Dle naměřených hodnot vytvořená testovací knihovna výkonnostně a funkčně splňuje podmínky aktuálně na ni kladené. S pomocí knihovny jsme schopní úspěšně orchestrovat automatizované testování v různých topologiích. Je zde ovšem stále prostor pro zlepšení. Z výkonnostního hlediska je teoreticky možné zrychlit vytváření a destrukci virtualizovaného prostředí, čímž by se zrychlil celý proces testování.

Menší nedostatky byly nalezeny i v použití knihovny. Aktuální omezení na operační systém Ubuntu snižuje možnosti jejího využití. Zároveň je zde možnost zlepšit registraci testů za pomoci sjednocení definice testu.

Celkově ovšem knihovna plní svojí funkci a zjednodušuje testování ve virtualizovaném prostředí.



Obrázek 7.4: Graf času potřebného k vytvoření virt. prostředí

Obrázek 7.5: Graf času potřebného k odstranění virt. prostředí

Závěr

Cílem této práce bylo analyzovat, navrhnout a implementovat rozšíření testovací knihovny o virtualizační prvky, které jí měly umožnit vytvářet zařízení ve virtualizovaném prostředí a propojovat je ve třech různých topologiích. Práce také měla demonstrovat způsob použití nové knihovny a zhodnotit její vlastnosti a možnosti reálného použití.

Cíle této práce byly splněny. Práce v kapitole 2 definuje jednotlivé pojmy a přibližuje kontext této práce. V kapitole 3 práce zhodnocuje stav předchozí knihovny a analyzuje možnosti jejího rozšíření. Následně v kapitole 4 práce navrhuje změny, dle výstupu z analýzy, jež jsou následně implementovány v kapitole 5. V kapitole 6 práce ukazuje možné použití knihovny s pomocí průmyslového protokolu ModbusTCP. V neposlední řadě práce zhodnocuje vlastnosti nové knihovny v kapitole 7.

Vytvořené řešení splňuje požadavky, které na něj byly kladeny, avšak jak již bylo zmíněno v kapitole 7, stále zde existuje prostor pro zlepšení. Knihovnu je ovšem ve svém aktuálním stavu možné využít při reálném vývoji.

Bibliografie

1. CAMPBELL, Sean; JERONIMO, Michael. An introduction to virtualization. *Published in "Applied Virtualization", Intel*. 2006, s. 1–15.
2. SUSANTA, Nanda; CHIUEH, Tzi-cker. A survey on virtualization technologies. *Rpe Report*. 2005, roč. 142.
3. RODRÍGUEZ-HARO, Fernando; FREITAG, Felix; NAVARRO, Leandro; HERNÁNCHEZ-SÁNCHEZ, Efraín; FARÍAS-MENDOZA, Nicandro; GUERRERO-IBÁÑEZ, Juan Antonio; GONZÁLEZ-POTES, Apolinar. A summary of virtualization techniques. *Procedia Technology*. 2012, roč. 3, s. 267–272. ISSN 2212-0173. Dostupné z DOI: https://doi.org/10.1016/j.protcy.2012.03.029. The 2012 Iberoamerican Conference on Electronics Engineering and Computer Science.
4. COMMONS, Wikimedia. *Privilege rings for the x86, along with their common uses*. [online]. 2007. [cit. 2023-04-10]. Dostupné z: https://commons.wikimedia.org/wiki/File:Priv_rings.svg. Soubor: Priv rings.svg.
5. WHITAKER, Andrew; SHAW, Marianne; GRIBBLE, Steven D et al. Denali: Lightweight virtual machines for distributed and networked applications. 2002.
6. ARM LTD. *What is Instruction Set Architecture (ISA)?* [online]. [B.r.]. [cit. 2023-03-14]. Dostupné z: https://www.arm.com/glossary/isa.
7. POPOVICI, Katalin; JERRAYA, Ahmed. Hardware Abstraction Layer. In: *Hardware-dependent Software: Principles and Practice*. Ed. ECKER, Wolfgang; MÜLLER, Wolfgang; DÖMER, Rainer. Dordrecht: Springer Netherlands, 2009, s. 67–94. ISBN 978-1-4020-9436-1. Dostupné z DOI: 10.1007/978-1-4020-9436-1_4.

8. FESL, Jan. *Virtualizace I - Virtualizační principy a technologie* [online]. 2023. [cit. 2023-04-10]. Dostupné z: `https://gitlab.fit.cvut.cz/NI-VCC/ni-vcc/tree/master/lectures/files/NI-VCC-Prednaska03.pdf`.
9. CHEN, Wei; LU, Hongyi; SHEN, Li; WANG, Zhiying; XIAO, Nong; CHEN, Dan. A Novel Hardware Assisted Full Virtualization Technique. In: *2008 The 9th International Conference for Young Computer Scientists*. 2008, s. 1292–1297. Dostupné z DOI: `10.1109/ICYCS.2008.218`.
10. SULTAN, Sari; AHMAD, Imtiaz; DIMITRIOU, Tassos. Container Security: Issues, Challenges, and the Road Ahead. *IEEE Access*. 2019, roč. 7, s. 52976–52996. Dostupné z DOI: `10.1109/ACCESS.2019.2911732`.
11. BENTALEB, Ouafa; BELLOUM, Adam S. Z.; SEBAA, Abderrazak; EL-MAOUHAB, Aouaouche. Containerization technologies: taxonomies, applications and challenges. *The Journal of Supercomputing*. 2021, roč. 78, č. 1, s. 1144–1181. Dostupné z DOI: `10.1007/s11227-021-03914-1`.
12. XAVIER, Miguel G.; NEVES, Marcelo V.; ROSSI, Fabio D.; FERRETO, Tiago C.; LANGE, Timoteo; DE ROSE, Cesar A. F. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2013, s. 233–240. Dostupné z DOI: `10.1109/PDP.2013.41`.
13. ALEKSIC, Marko. *10 Docker Alternatives* [online]. 2023. [cit. 2023-02-19]. Dostupné z: `https://phoenixnap.com/kb/docker-alternatives`.
14. BELLARD, Fabrice. QEMU, a fast and portable dynamic translator. In: *USENIX annual technical conference, FREENIX Track*. Califor-nia, USA, 2005, sv. 41, s. 46.
15. GILLIS, Alexander S. *What is VMware ESXi? - Definition from TechTarget.com* [online]. TechTarget, 2022 [cit. 2023-04-25]. Dostupné z: `https://www.techtarget.com/searchvmware/definition/VMware-ESXi`.
16. LI, Ziyu. Comparison between common virtualization solutions: VMware Workstation, Hyper-V and Docker. In: *2021 IEEE 3rd International Conference on Frontiers Technology of Information and Computer (ICFTIC)*. 2021, s. 701–707. Dostupné z DOI: `10.1109/ICFTIC54370.2021.9647226`.
17. TURNBULL, James. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.
18. *Docker overview* [online]. [B.r.]. [cit. 2023-02-20]. Dostupné z: `https://docs.docker.com/get-started/overview/`.
19. DOCKER INC. *Docker Hub* [online]. [B.r.]. [cit. 2023-04-25]. Dostupné z: `https://hub.docker.com/`.

20. *Networking overview* [online]. [B.r.]. [cit. 2023-02-23]. Dostupné z: https://docs.docker.com/network/.

21. *Dockerfile reference* [online]. [B.r.]. [cit. 2023-04-30]. Dostupné z: https://docs.docker.com/engine/reference/builder/.

22. *Docker Compose overview* [online]. [B.r.]. [cit. 2023-04-30]. Dostupné z: https://docs.docker.com/compose/.

23. *What is Podman?* [online]. [B.r.]. [cit. 2023-02-25]. Dostupné z: https://docs.podman.io/en/latest/.

24. *Podman vs Docker: All You Need To Know!* [online]. 2022. [cit. 2023-02-25]. Dostupné z: https://www.lambdatest.com/blog/podman-vs-docker/.

25. ŠTĚPÁNEK, Martin. *Návrh a implementace knihovny pro automatizaci testů verifikace průmyslové komunikace*. 2021. Bakalářská práce. České vysoké učení technické v Praze.

26. *Features/KVM* [online]. [B.r.]. [cit. 2023-05-02]. Dostupné z: https://wiki.qemu.org/Features/KVM.

27. CHAE, MinSu; LEE, HwaMin; LEE, Kiyeol. A performance comparison of linux containers and virtual machines using Docker and KVM. *Cluster Computing*. 2019, roč. 22, č. Suppl 1, s. 1765–1775.

28. BHARDWAJ, Aditya; KRISHNA, C Rama. Virtualization in cloud computing: Moving from hypervisor to containerization—a survey. *Arabian Journal for Science and Engineering*. 2021, roč. 46, č. 9, s. 8585–8601.

29. *Use bridge networks* [online]. [B.r.]. [cit. 2023-02-23]. Dostupné z: https://docs.docker.com/network/bridge/.

30. *Container networking* [online]. [B.r.]. [cit. 2023-04-30]. Dostupné z: https://docs.docker.com/config/containers/container-networking/.

31. MICROSOFT CORPORATION. *Nuget Gallery: Home* [online]. [B.r.]. [cit. 2023-04-03]. Dostupné z: https://www.nuget.org/.

32. .NET FOUNDATION. *Dotnet/docker.dotnet: .NET (C#) client library for docker API* [online]. [B.r.]. [cit. 2023-04-03]. Dostupné z: https://github.com/dotnet/Docker.DotNet/.

33. TOFFIA, Mario. *Mariotoffia/FluentDocker: Use Docker, Docker-compose local and remote in tests and your .NET core/full framework apps via a fluentapi* [online]. [B.r.]. [cit. 2023-04-03]. Dostupné z: https://github.com/mariotoffia/FluentDocker.

34. .NET FOUNDATION. *Who we are?* [online]. [B.r.]. [cit. 2023-04-03]. Dostupné z: https://dotnetfoundation.org/about/who-we-are.

35. FOWLER, Martin. *FluentInterface* [online]. 2005. [cit. 2023-04-03]. Dostupné z: `https://martinfowler.com/bliki/FluentInterface.html`.

36. MAREŠ, Martin; VALLA, Tomáš. *Průvodce labyrintem algoritmů*. CZ. NIC, 2022.

37. DOTPCAP. *Dotpcap/sharppcap: Official Repository - fully managed, cross platform (Windows, mac, linux) .NET library for capturing packets* [online]. [B.r.]. [cit. 2023-04-04]. Dostupné z: `https://github.com/dotpcap/sharppcap`.

38. MODBUS ORGANIZATION. *Modbus Application Protocol Specification v1.1b3* [online]. 2012. [cit. 2023-05-01]. Dostupné z: `https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf`.

39. *Sourceperl/pyModbusTCP: A simple modbus/TCP library for python* [online]. [B.r.]. [cit. 2023-05-01]. Dostupné z: `https://github.com/sourceperl/pyModbusTCP`.

40. MICROSOFT CORPORATION. *Visual studio 2022 17.5.1* [Software]. 2023. [cit. 2023-04-04]. Dostupné z: `https://visualstudio.microsoft.com/vs/`.

Příloha A

Seznam použitých zkratek

API Application Programming Interface

BFS Breadth First Search

I/O Input/Output

IDE Integrated Development Environment

ISA Instruction Set Architecture

JSON JavaScript Object Notation

HAL Hardware Abstraction Layer

HPC High performance counting

MB/s Megabyte za sekundu

OCI Open Container Initiative

OS Operační systém

PLC Programmable Logic Controller

QA Quality assessment

REST Representational State Transfer

VM Virtual machine

YAML YAML Ain't Markup Language

Příloha B

Obsah přílohy

- `README.txt` ..... stručný popis obsahu SD karty
- `impl` ..... zdrojové kódy implementace
  - `base_docker_image` ..... výchozí Docker obraz
  - `core` ..... implementace jádra testovací knihovny v jazyku C#
  - `cpp` ..... implementace jádra testovací knihovny v jazyku C++
  - `python` ..... implementace jádra testovací knihovny v jazyku Python
  - `test_lib` ..... implementace testovací knihovny v jazyku C#
  - `test_project` ... testovací projekt vytvořený k demonstraci funkčnosti
- `package` .... adresář se zkompilovanými NuGet balíčky testovací knihovny
- `text` ..... text práce
  - `src` ..... zdrojová forma práce ve formátu LaTeX
  - `thesis.pdf` ..... text práce ve formátu PDF