FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Návrh a implementace knihovny pro automatizaci testů verifikace průmyslové komunikace
Student:	Martin Štěpánek
Vedoucí:	Ing. Miroslav Dušek
Studijní program:	Informatika
Obor / specializace:	Informační systémy a management
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Cílem práce je vytvoření knihovny na automatizování testů pro ET200SP IO systém, který je založen na Multifieldbus technologii, která kombinuje několik průmyslových komunikačních protokolů.
1. Navrhněte knihovnu pro automatizaci testů, což znamená:
- Navrhněte službu, která bude řídit provedení testu na testovaném zařízení.
- Navrhněte rozhraní, které umožnuje implementaci knihovny na testovaném zařízení.
- Navrhněte protokol pro komunikaci mezi testovaným zařízením a testovací službou.
2. Navrhněte rozhraní pro propojení testovací služby a Azure DevOps serverem tak, aby Azure DevOps server mohl automaticky spouštět testy implementované na testovaném zařízení.
3. Prozkoumejte existující open-source knihovny pro protokoly ModbusTCP a Ethernet/IP a vyberte kandidáty na implementaci do knihovny.
4. Implementujte základní testy pro jeden průmyslový protokol jako demonstraci funkcionality vašeho řešení.
5. Zhodnoťte výsledné řešení z pohledu projektového řízení.

Elektronicky schválil/a Ing. David Buchtela, Ph.D. dne 2. února 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Návrh a implementace knihovny pro automatizaci testů verifikace průmyslové komunikace

Martin Štěpánek

Katedra softwarového inženýrství
Vedoucí práce: Ing. Miroslav Dušek

13. května 2021

Poděkování

Rád bych poděkoval Ing. Miroslavu Duškovi za vedení práce a veškerou pomoc. Zároveň bych chtěl poděkovat ostatním kolegům ve firmě Siemens, s. r. o., především Ing. Borisi Bulánkovi, za cené rady během vývoje testovací knihovny. 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 13. května 2021 .....

Odkaz na tuto práci

Štěpánek, Martin. *Návrh a implementace knihovny pro automatizaci testů verifikace průmyslové komunikace*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací knihovny pro automatizaci testů verifikace průmyslové komunikace. Vytvořená knihovna funguje jako doplněk do testovacího frameworku MSTest, za pomocí něhož je knihovna následně propojena se serverem Azure DevOps. Ten následně může jednotlivé testy registrovat a automaticky spouštět. Práce také ukazuje open-source knihovny pro průmyslové protokoly ModbusTCP a EtherNet/IP, které jsou vhodné k využití současně s testovací knihovnou, a za pomocí jedné z těchto knihoven následně demonstruje funčknost vytvořeného řešení. Na závěr práce zhodnocuje vytvořené řešení a jeho přínos.

Klíčová slova softwarové testování, automatizace testů, testovací knihovna, verifikace průmyslové komunikace, MSTest, ModbusTCP, EtherNet/IP

Abstract

This bachelor's thesis is focused on design and implementation of test automation library for verification of industry fieldbus communication. This library works as add-in to testing framework MSTest, which enables connection with Azure DevOps server. Azure DevOps server is then able to register each of the tests and automatically execute them. This thesis also shows open-source libraries for fieldbuses ModbusTCP and EtherNet/IP, which are suitable for use together with the created test library. Thesis also demonstrates the funcionality of the created test library, with the help of one of the open-source library for fieldbus. Lastly the thesis evalutes the created solution and its contribution.

Keywords software testing, test automation, test library, verification of fieldbus communication, MSTest, ModbusTCP, EtherNet/IP

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Jedním z cílů průmyslu je již od jeho vzniku zefektivnění výroby, které poté vede ke zvýšení zisků. V průběhu historie průmyslu se každá velká změna označuje za tzv. revoluci. V dnešní době se nacházíme ve čtvrté revoluci, která je často označována jako Průmysl 4.0.

Cílem této revoluce je ještě větší automatizace opakujících se činností, kterou vykonávají lidé, digitalizace a zefektivnění komunikace mezi všemi zařízeními. Toto je ještě více podstatné v kontextu dnešní doby, kdy ve světě řádí pandemie, a výrobci si ještě více uvědomují křehkost lidské pracovní síly. Díky tomu jsou kladeny mnohem větší požadavky na komunikaci v průmyslových sítích, v nichž jsou používány specializované průmyslové protokoly.

Při vývoji zařízení, které se podílí na automatizaci výroby, je stejně důležité, jako například jejich návrh, rovněž jejich testování. Hlavním úkolem této fáze je odhalení nedostatků produktu, které se liší od dané specifikace produktu. Testováním je tedy produkt kontrolován a z výsledků testů lze odvodit stav a kvalitu produktu. V dnešní době se u testování snaží využít výhod automatizace testování. Mezi tyto výhody patří například jednoduchá opakovatelnost testování nebo umožnění častějšího testování.

Tato práce se věnuje návrhu a implementaci knihovny, která bude automatizovat testy verifikace průmyslové komunikace. Hlavní motivací k vytvoření této knihovny je standardizace testování, zjednodušení a zrychlení vytváření testů. Toto poté vede k zjednodušení testování a snížení nákladů na testování. Tato knihovna je vytvářena pro společnost Siemens, s. r. o.

Práce začíná kapitolou 1, ve které stanovuje cíle práce. V kapitole 2 práce definuje jednotlivé pojmy používané v práci a přibližuje kontext této práce. Následně kapitola 3 se věnuje návrhu testovací knihovny a všech jejich komponent. Implementace všech komponent je popsána v kapitole 4. Použití vytvořeného řešení je popsáno v kapitole 5. V neposlední řadě kapitola 6 se věnuje zhodnocení vytvořeného řešení a jeho přínosu.

Kapitola 1

Cíl práce

Cílem této práce je navrhnout a implementovat knihovnu, která umožní automatizovat testy verifikace průmyslové komunikace. Součástí této knihovny má být:

- služba, která bude řídit testovací běh,
- rozhraní, které umožní implementaci knihovny na testovaném zařízení,
- protokol, který bude definovat komunikaci mezi službou a testovanými zařízeními.

Vytvořená knihovna má být poté propojena s Azure DevOps serverem tak, aby Azure DevOps server mohl následně automaticky spouštět testy. Následně má server Azure DevOps obdržet výsledky těchto testů a poskytnout uživateli informaci o jejich průběhu.

Dalším cílem je provést výzkum dostupných open-source knihoven pro průmyslové protokoly ModbusTCP a EtherNet/IP a vybrat vhodné kandidáty na implementaci do této knihovny.

Součástí této práce má též být demonstrace použití vytvořené testovací knihovny, kde bude využit jeden z průmyslových protokolů. V neposlední řadě je cílem zhodnotit vytvořené řešení a jeho přínosy z pohledu projektového řízení.

Kapitola 2

Teoretická část

V této kapitole se věnuji přiblížení teoretického kontextu této práce.

2.1 Testování

Testování je podstatnou součástí vývoje produktu a jeho softwaru. Cílem testování není pouze odhalení chyb v softwaru, ale také verifikace a validace softwaru. Validací softwaru kontrolujeme, zda software odpovídá dané specifikaci produktového managmentu či konkterétního zákazníka a je tím, co zákazník chtěl. Verifikací kontrolujeme správnost softwaru, tedy kontrolujeme, že systém ve vytvořených situacích se chová dle očekávání a specifikace. [1]

Testování softwaru je zároveň dovednost. Při testování musí tester vybrat z nekonečného množství možných testů jejich konečný počet, který nejlépe reprezentuje danou problematiku a pokrývá co největší možnou množinu všech možných případů. Zároveň musí vzít v potaz náročnost na vytvoření testu a na rigidnost vytvořeného testu proti změnám v softwaru. Tyto faktory poté ovlivňují i náklady na testování. [2]

Od testování softwaru lze rovněž odvodit i jeho kvalitu. Ta se dá určit tím, jak moc vytvořený software odpovídá zadaným specifikacím [3]. Tyto informace jsou poté velmi důležité pro management. Díky nim může vyhodnocovat současný stav vývoje a upravovat plán na vývoj.

Testování zároveň zvyšuje důvěru ve vyvíjený software. Každý dobře navržený test snižuje šanci, že v softwaru existuje nepodchycená chyba. S každým rozsáhlým testováním se tato důvěra zvyšuje. [2]

2.1.1 Rozdělení testů

I když primární cíl testování je jednotný, přístupů k testování je několik. Vhodnost jednotlivých přístupů se mění na základě testované komponenty. Tyto přístupy se dají podle [4] rozdělit do několika kategorií.

2.1.1.1 Podle znalosti komponenty

Testování se dá rozdělit podle přístupu k informacím, které o komponentách softwaru/systému víme. Patří sem:

Black box testování Nazýváno taktéž funkční testování. Na software se pohlíží jako na tzv. černou skříňku. O komponentě nebo celku nic nevíme a testujeme na základě funkcionálních požadavků, návrhu, specifikací nebo uživatelské dokumentace.

White box testování Se znalostí implementace testované části se snažíme vytvořit takové testy, které způsobí spouštění určitých částí testované komponenty. Cílem je co největší pokrytí testování dané komponenty.

Grey box testování Kombinace Black box a White box testování. Při testování máme nějakou znalost implementace komponenty, ale je nižší, než při White box testování. [5]

2.1.1.2 Podle částí vývoje

Testování podle částí vývoje se přibližuje vývojovému cyklu. Do těchto kategorií patří:

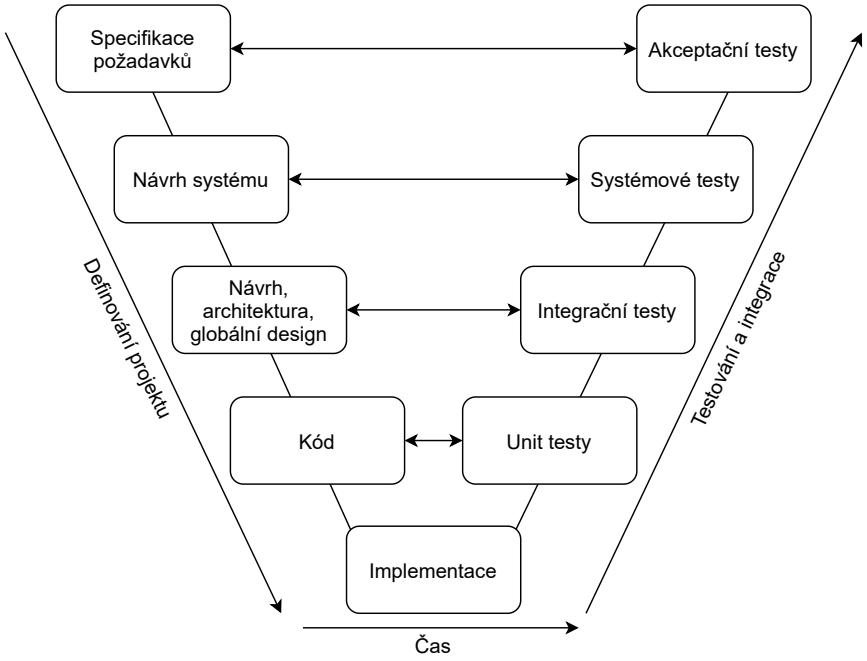
Testování jednotlivých částí Více známé pod anglickým pojmem „Unit testing“. Je to nejnižší úroveň testování. Testuje jednotlivé komponenty systému samostatně.

Integrační testování Testování dvou a více komponent, které spolu vytváří nějaký větší celek softwaru. Často také využíván při testování částí, které nelze samostatně testovat.

Systémové testování Testování softwaru jako celku. Zaměřuje se na testování funkčních požadavků. Zároveň je možno vyhodnocovat další požadavky na systém, jako spolehlivost, bezpečnost, atd.

Akceptační testování U toho testování se systém dostane do rukou zákazníkovi/uživatelům. Cílem je otestování produktu u uživatelů softwaru a získání jejich zpětné vazby.

Propojení testů a vývojového cyklu je dobře znázorněno na tzv. V-modelu, který můžeme vidět na obrázku 2.1. Na tomto modelu, nazývaném podle svého tvaru, můžeme vidět jednotlivé typy stádia vývoje a jejich korespondující ověřování v závislosti na čase.



Obrázek 2.1: V-model

Zdroj: Vytvořeno dle předlohy z [2]

2.2 Automatizace testování

Automatizace testování je odlišná od samotného testování. Automatizace testu neurčuje jeho samotnou kvalitu. Pokud automatizujeme test, který nepřináší žádné nové benefity vývoji softwaru, tak poté dostaneme jeho výsledek pouze rychleji. [2] Proto je tvorba testů minimálně stejně důležitá, ne-li důležitější, jako samotná jejich automatizace. Automatizování je ale v mnoha ohledech v dnešní době standardem při testování, a to především díky jeho výhodám. Mezi tyto výhody podle [2] patří:

Častější testování S automatizací jsme schopni testovat software mnohem častěji, než při manuálním testování. Software může být testován například při každé jeho změně. Toto je manuálně velmi náročné, už jen kvůli vysokému požadavku na lidské zdroje.

Ulehčení testování Test, který bude požadovat k jeho uskutečnění 200 uživatelů, není nemožný provést manuálně. S automatizací můžeme vstupy těchto uživatelů simulovat a tím celé testování zjednodušit.

Lepší využití zdrojů Tester je kvalifikovaný člověk a jeho využití na opakované vkládání vstupů a ověřování výstupů může být v některých případech plýtváním jeho časem. Taktéž při manuálním testování můžeme

vyžadovat více účastníků testování, ale všichni si musí vyhradit čas na testování. Díky automatizaci můžeme tyto zdroje využít mnohem efektivněji. Tester se například díky automatizaci může více soustředit na vytváření nových testů.

Konzistence Při automatizaci testování každý běh testu proběhne naprosto identicky. Stejně jako ve vývoji, i v testování může dojít k lidské chybě. Díky automatizaci se šance lidské chyby snižuje. Toto zvyšuje konzistenci testování, než když se testy provádějí manuálně.

Snížení doby testování Jednou automatizované testy mohou být provedeny mnohem rychleji a efektivněji, než při jejich manuálním spuštění. Toto způsobuje snížení potřebné doby na testování.

Automatizované testování následně může podporovat další procesy během vývoje. Mezi tyto procesy patří například proces kontinuální integrace nebo proces kontinuálního doručení softwaru. Proces kontinuální integrace se zaměřuje na integrování jednotlivých upravovaných částí do celkového softwaru. Oproti tomu proces kontinuálního doručení softwaru se zaměřuje na to, aby software vždy byl v takovém stavu, aby mohl být doručen zákazníkovi, nebo nasazen. [6]

2.3 Azure DevOps

Azure DevOps server poskytuje vývojářům služby, které pomáhají při vývoji softwaru. Jeho cílem je podporovat jednotlivé procesy vývoje, což poté zrychluje vývoj softwaru. [7] V této práci bude využito několika služeb, které Azure DevOps server nabízí. Tyto služby budou:

Azure Repos Azure Repos je sada nástrojů, která umožňuje správu jednotlivých verzí softwaru. V této práci budeme používat systém správy verzí Git, který je službou Azure Repos podporovaný. [8]

Azure Pipelines Azure Pipelines umožňuje automaticky kompilovat a testovat vyvíjený software. Tím také podporuje procesy kontinuální integrace, kontinuálního nasazení softwaru nebo kontinuálního testování. [9]

Azure Test Plans Azure Test Plans přináší sadu nástrojů, které umožňují spravovat testování softwaru. Za pomocí těchto nástrojů lze spravovat testovací sady a jednotlivé testy, ať už manuální nebo automatizované. [10]

Azure Artifacts Azure Artifacts umožňuje publikování a verzování různých typů balíčků. Následně může být využita při vydání těchto balíčků. Mezi podporované balíčky patří například NuGet, Maven nebo npm. Zároveň skrze Azure Artifact můžeme publikovat data z Azure Pipelines. [11]

Azure DevOps server využívá při delegování úkolů tzv. agenty. Agent je výpočetní infrastruktura s nainstalovaným softwarem agenta, který pracuje na jedné určité úloze [12]. Agent následně provádí například jednotlivé úkony definované v Azure Pipelines, jako kompilace, testování atd.

2.4 Framework MSTest

Framework MSTest je výchozí testovací framework, který je integrován do IDE Visual Studio. Díky tomu je také často nazýván jako „Visual Studio Unit Testing Framework“. Framework započal jako nástroj spouštěný z příkazové řádky, který následně prováděl testování. Díky implementaci do IDE Visual Studio je tento framework často preferován vývojáři, kteří používají toto IDE pro vývoj.

MSTest framework přináší nástroje, které jsou potřeba k verifikaci a validaci softwaru. V dnešní době framework MSTest V2 je open-source projekt, který je stále rozvíjen. Mezi jeho výhody patří podpora napříč platformami a rozšiřitelnost. [13]

2.5 Průmyslová komunikace

Při řešení průmyslové komunikace se často objevuje slovo *fieldbus*. Běžný význam tohoto slova je „Síť, která propojuje průmyslová zařízení jako kontrolery, PLC, regulátory atd.“ [14]. Vznik těchto sítí je úzce spojený s historií vývoje informačních technologií. V době, kdy začaly tyto sítě vznikat, nebyly dostupné komunikační technologie jako dnes. Dostupné informační a telekomunikační sítě té doby nemohly uspokojit potřeby průmyslových sítí na deterministickou, spolehlivou a efektivní komunikaci [15].

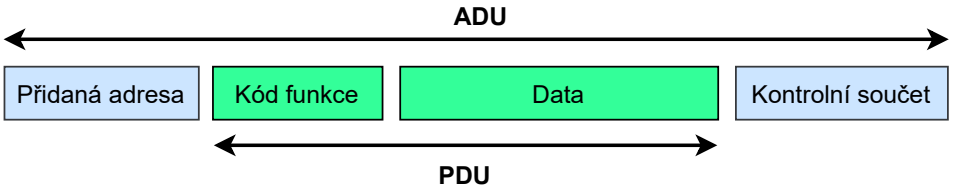
V dobách vzniku těchto komunikačních protokolů bylo pro fyzické připojení využíváno různých protokolů – například RS-485. Nedostatkem těchto protokolů však byly vysoké implementační náklady. S růstem popularity standardu IEEE 802.3 – dnes často nazýván Ethernet – se začalo zkoumat možné použití tohoto standardu při průmyslové komunikaci. Odborníci došli k závěru, že je možné při průmyslové komunikaci v reálném čase využít tento standard [16]. V dnešní době je tento standard hojně využíván při přenosu průmyslové komunikace.

2.5.1 Protokol Modbus

Modbus je průmyslový komunikační protokol, který je umístěn na aplikační vrstvě ISO/OSI modelu. Tento protokol, vytvořený v roce 1979, umožňuje komunikaci typu klient-server mezi zařízeními na různých typech sítí a sběrnic.

Protokol definuje strukturu zprávy (tzv. PDU – Protocol Data Unit) nezávisle od komunikační vrstvy. Tato zpráva se však může rozšířit v závislosti

na způsobu přenosu této zprávy. V závislosti na typu sítě je poté tato zpráva rozšířena o další údaje. Tento celek se označuje jako ADU – Application Data Unit. Celou strukturu můžeme vidět znázorněnou na obrázku 2.2. [17]



Obrázek 2.2: Obecné znázornění jednoho rámce protokolu Modbus

Zdroj: Vytvořeno dle předlohy z [17]

Podporované služby jsou definovány funkčními kódy, které jsou součástí požadavku. Kód funkce je uložen v jednom bajtu a tedy rozsah možných kódů je 1-255 (0 je nevalidní), kde rozsah 128-255 je rezervován pro chybové zprávy. Data zprávy poté slouží k upřesnění operace definované funkčním kódem. V určitých situacích muže postačovat v provedení operace funkční kód, data zprávy poté mají nulovou délku.

Modbus používá komunikaci požadavek-odpověď. Na požadavek klienta server odpovídá zprávou, kterou značí úspěch či neúspěch operace. V případě úspěchu server odesílá zpět zprávu se stejným funkčním kódem, jako byl požadavek. V opačném případě odesílá zpět zprávu, která obsahuje stejný kód funkce, jako byl kód funkce požadavku, ale tento kód má zároveň nastavený nejvýznamnější bit na 1. Zpráva v obou případech může poté obsahovat data, v závislosti na operacích. [17]

Tato práce se bude zaměřovat na použití protokolu ModbusTCP. Jak již z názvu vyplývá, ModbusTCP používá k přenosu zpráv protokol TCP/IP a Ethernet připojení. Zároveň používá stejnou komunikaci definovanou protokolem Modbus.

2.5.2 Protokol EtherNet/IP

Protokol EtherNet/IP (EtherNet/Industrial Protocol) je dalším, dnes již široce používaným průmyslovým protokolem. Jak již z názvu vyplývá, protokol používá Ethernetový standard k propojení průmyslových zařízení. Ke komunikaci následně používá standardně používané protokoly TCP a UDP. Tento protokol byl poprvé představen v roce 2001 a od roku 2005 je standardizován.

Protokol je umístěn na 5–7 vrstvě ISO/OSI modelu. V rámci sítě jsou jednotlivým uzlům přiřazeny předem definované profily. Profil reprezentuje typ zařízení se specifickými vlastnostmi. Profil zařízení a aplikační vrstva protokolu EtherNet/IP jsou vytvářeny protokolem CIP (Common Industrial Protocol). Tento protokol používá komunikaci na principu producent-konzument.

Použitím společného protokolu CIP se následně dosahuje interoperability mezi všemi sítěmi, které ho podporují.

Protokol CIP je objektově orientovaný. Každé zařízení je reprezentováno skupinou objektů. Každý z nich obsahuje data, služby, resp. příkazy a specifikaci funkcí. V rámci protokolu je poté definováno, jaké atributy musí každý objekt obsahovat.

Aplikační objekty obsahují data, která jsou specifická pro komunikující zařízení. Výrobci mohou specifikovat i svoje vlastní objekty. Pro identifikaci dostupných objektů jsou sestaveny elektronické popisy zařízení, které obsahují potřebné informace ke konfiguraci zařízení v síti EtherNet/IP.

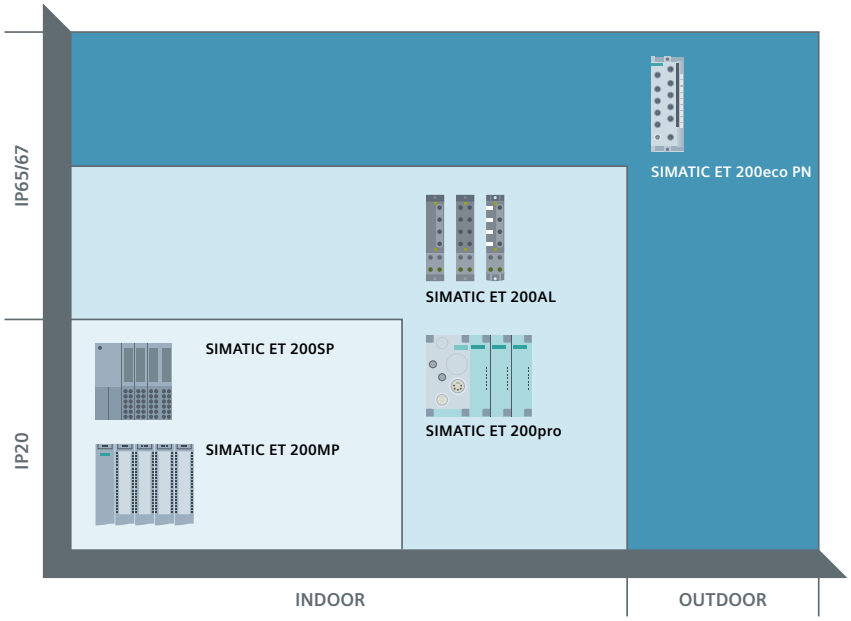
Přenos v síti EtherNet/IP se odvíjí dle využitého protokolu pro přenos. Při použití protokolu TCP je přenos explicitní a je určen k přenosu typu žádost-odpověď mezi dvěma zařízeními. Naopak při použití protokolu UDP je přenos implicitní, který je pak určený pro cyklický přenos uživatelských a I/O dat. Komunikační model objektů CIP umožňuje lépe využít možnosti komunikačního kanálu. Přenos zprávy následně probíhá dle protokolu. Žadatel zahajuje komunikaci s cílovým zařízením odesláním žádosti k vytvoření komunikace, která obsahuje navrhované parametry spojení. Cílové zařízení následně odesílá potvrzení s přesnými parametry a navazuje spojení. Každé spojení je poté určeno identifikátorem. [18]

2.6 Testovaný produkt

Hlavním cílem této práce je vytvoření testovací knihovny pro řadu zařízení SIMATIC ET 200, vyvíjený společností Siemens, s. r. o., se zaměřením na zařízení SIMATIC ET 200SP. Řada produktů SIMATIC ET 200 přináší jednu z nejširších nabídek I/O zařízení. Produkty této řady můžeme vidět znázorněné na obrázku 2.3. Jak je z obrázku patrné, každé zařízení obsahuje jiný stupeň ochrany v závislosti na jejich plánovaném umístění. [19]

Zařízení SIMATIC ET 200SP, které můžeme vidět na obrázku 2.4, je určeno k umístění do kompaktních řídících kabinetů. Díky modulárnosti systému se systém může vysoce přizpůsobit potřebám jednotlivých průmyslů. Mezi tyto moduly patří standardní I/O moduly, komunikační moduly, technologické moduly nebo moduly ke startování motorů. Zároveň je velikou výhodou celkový kompaktní design. Jednotlivé moduly zároveň mohou být vyměňovány za chodu. [20]

SIMATIC ET 200SP využívá tzv. *Multifieldbus technologii*. Tato technologie umožňuje zařízení komunikovat na základě několika průmyslových protokolů. V rámci této práce se zaměříme na protokoly ModbusTCP a EtherNet/IP, které byly popsány již dříve.



Obrázek 2.3: Znázornění produktové řady SIMATIC ET 200

Zdroj: Marketingové materiály firmy Siemens, s. r. o. [19]

Obrázek 2.4: Zařízení SIMATIC ET 200SP

Zdroj: Marketingové materiály firmy Siemens, s. r. o. [20]

Kapitola 3

Návrh

V této kapitole se zabývám návrhem testovací knihovny a jejími funkčními požadavky.

3.1 Účastníci testování

V rámci knihovny a testovacího běhu se bude vyskytovat několik účastníků testování. Mezi účastníky testování bude patřit:

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.

Testovací služba bude jádrem k řízení testování. Poběží na samostatném zařízení, které bude běžet na operačním systému Windows. Zařízení bude v rámci infrastruktury serveru Azure DevOps agent. Testovací služba bude implementována v jazyce C#. Všechna implementace v jazyce C# bude vytvořena pro .NET Framework 4.8.

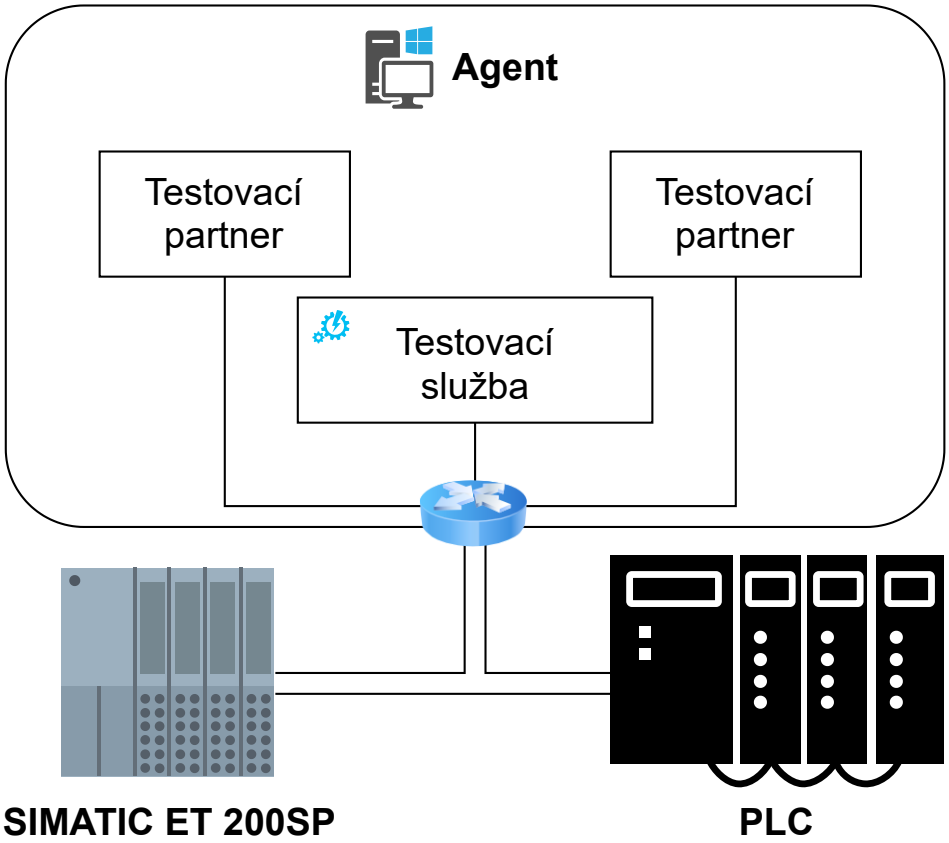
Hlavním cílem je testovat vyvíjený produkt. Toto zařízení bude v rámci knihovny bráno jako testované zařízení. Testovací služba bude podporovat připojení jednoho a více testovaných zařízení a při každém testovacím běhu musí být připojeno alespoň jedno testovací zařízení. Tato zařízení budou připojena k zařízení, ze kterého poběží testovací služba, za pomoci Ethernet připojení. Zařízení SIMATIC ET 200SP má svoji implementaci v jazyce C++. Z toho vyplývá, že implementace pro toto zařízení bude vytvořena v tomto jazyce.

Součástí knihovny zároveň bude také tzv. testovací partner. Tento účastník slouží k simulaci testovaného zařízení, buď jako celku, nebo pouze nějaké

jeho činnosti. Zařízení může simulovat zařízení jako PLC, nebo různé nástroje, které slouží například k certifikaci vyvíjeného zařízení. Simulováním zařízení snižujeme hardwarové nároky na testování, což vede ke snížení ekonomických nákladů na testování. Tato zařízení se budou připojovat v závislosti na jednotlivých testech. Následně po skončení testu budou tato zařízení odpojena. Testovací zařízení bude stejně jako testovací služba vytvořeno v jazyce C#.

Testovací služba bude očekávat připojení minimálně jednoho testovaného zařízení. Následně maximální počet připojených zařízení ke službě se bude odvíjet od limitací protokolu TCP/IP a jeho implementace v jazyce C#. Tento limit se ale pohybuje ve stovkách, v kontrastu s tím se očekávaný počet zároveň připojených zařízení bude pohybovat v desítkách. Tedy tento limit knihovnu nijak neomezuje.

Účastníky testování a jejich možné propojení znázorňuje obrázek 3.1. Zde můžeme vidět agenta, resp. zařízení, na kterém poběží testovací služba a případní testovací partneři, primární testované zařízení SIMATIC ET 200SP a PLC. PLC zařízení značí jakékoliv zařízení, které není simulováno knihovnou.

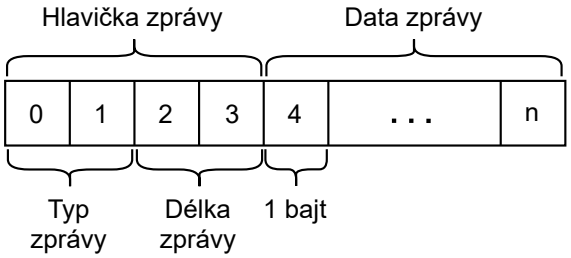


Obrázek 3.1: Ukázka možného propojení účastníků testování

3.2 Komunikace

Komunikace se všemi účastníky testování bude fungovat na principu TCP/IP připojení. Testovací služba a zařízení, na kterém služba poběží, bude sloužit jako server a všichni účastníci testovaní budou klienti, kteří se budou k tomuto zařízení připojovat.

Jednotlivé zprávy vyměňované mezi testovací službou a všemi účastníky testování budou mít jasně stanovenou strukturu. Diagram složení zprávy lze vidět na obrázku 3.2, z něhož je patrné, že jedna zpráva lze rozdělit na dvě části – hlavičku zprávy a data zprávy. Zároveň, jak můžeme vidět, jedna buňka v diagramu odpovídá jednomu bajtu.



Obrázek 3.2: Diagram struktury jedné zprávy

Zpráva bude povinně obsahovat hlavičku zprávy, kde bude uveden typ zprávy a délka dat zprávy. Jak můžeme vidět na diagramu, každý z těchto údajů bude mít velikost dva bajty. Celkově tedy hlavička bude o velikosti 4 bajty. Knihovna bude podporovat tyto typy zpráv:

1. `MSG_OK` – Zpráva o úspěchu/potvrzení
2. `MSG_FAIL` – Zpráva o neúspěchu
3. `MSG_TEST` – Direktiva ke spuštění testu
4. `MSG_STOP` – Direktiva k ukončení testování

Za hlavičkou zprávy následně budou moct být uložena data zprávy. Tato data však budou nepovinná. V těchto případech, kdy zpráva nebude neobsahovat data zprávy, bude pro přenesení informace postačovat pouze typ zprávy a v hlavičce bude uvedena nulová hodnota jako délka dat zprávy.

Maximální délka jedné zprávy se bude odvíjet od limitací použitých technologií. Limit se bude primárně odvíjet od Ethernet protokolu, který podporuje nejmenší maximální délku jednoho paketu ze všech použitých protokolů, a to 1500 bajtů. Toto číslo také zahrnuje všechny potřebné hlavičky protokolu TCP/IP k přenosu paketu. Z toho důvodu bude jako maximální délka jedné zprávy zvolena délka 1400 bajtů. Toto číslo ponechává místo pro potřebné hlavičky a zároveň je více než dostatečné pro navrhované použití. [21]

Všechny hodnoty zprávy budou ukládány v kladné (anglicky tzv. unsigned) podobě. Jedinou kontrolní informací, kterou bude zpráva obsahovat, je délka dat zprávy. Protokol TCP/IP sám obsahuje jednoduchou detekci chyb. Dohromady tyto kontroly budou považovány jako dostatečné.

Účastníci testování se na začátku připojí k testovací službě a odešlou inicializační zprávu. Tato zpráva bude typu 1 a v datech zprávy bude obsahovat svou MAC adresu. Výjimkou budou testovací partneři, kteří budou jako svoji MAC adresu odesílat adresu `DE:AD:BE:EF:00:00`. Díky tomu je testovací služba jednoduše identifikuje. Testovací služba následně odešle potvrzovací zprávu o úspěšném připojení s typem 1, bez žádných dat. V případě vzniku chyby služba odešle zprávu typu 2, bez dat.

Pro započetí testování a spuštění jednotlivých stádií testování služba odešle všem účastníkům testování zprávu s typem 3. V datech zprávy služba odesílá tato data:

1. Číselnou reprezentaci identifikátoru testu. Ten je uložen v prvních 4 bajtech dat zprávy.

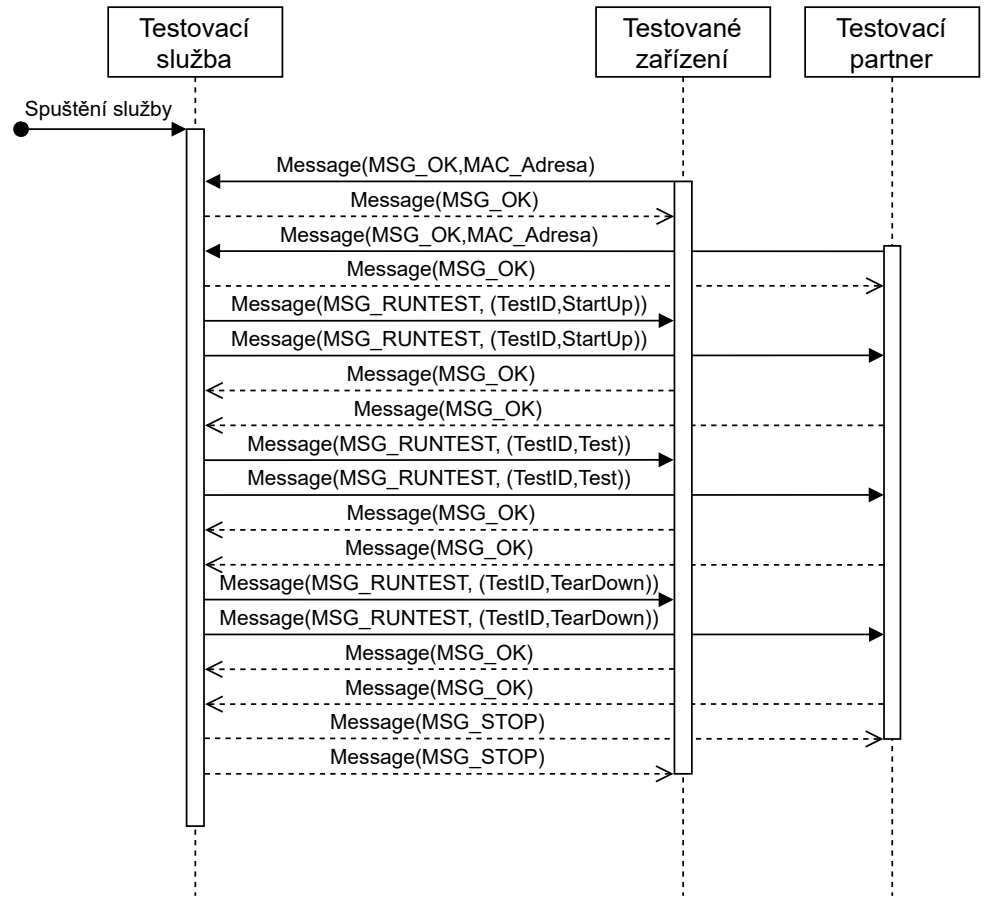
2. Číselnou reprezentaci identifikátoru, který určuje fázi testu. Uložen je ve dvou bajtech dat zprávy hned za identifikátorem testu. Fáze testování jsou blíže popsány v sekci 3.3.4.

Služba následně čeká na odpověď od všech účastníků testování. Účastnící odesílají testovací službě po skončení testovacího stádia zprávu bez dat, s typem 1 v případě úspěchu a s typem 2 v případě neúspěchu. Po skončení testování služba odesílá všem účastníkům testování zprávu s typem 4, bez žádných dat.

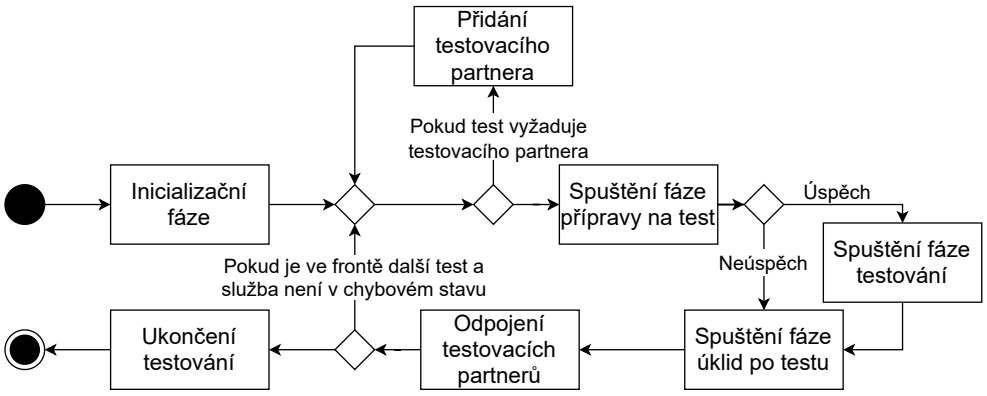
Ukázka komunikace je znázorněna na sekvenčním diagramu, viz obrázek 3.3. Na tomto sekvenčním diagramu můžeme vidět výměnu zpráv mezi testovací službou a dvěma účastníky testování během spuštění jednoho úspěšného testu. Direktiva `Message` znázorňuje jednu zprávu, kde v závorkách můžeme vidět nejdříve typ zprávy a následně data zprávy (pokud zpráva nějaká data obsahuje). Rovněž lze v diagramu spatřit využití testovacího partnera.

3.3 Testovací služba

Jak jsem již zmínil v sekci 3.1, testovací služba bude jádrem celého testování. Tato služba bude řídit celý testovací běh a předávat všem účastníkům testování pokyny. Zároveň bude synchronizovat testovací běh mezi všemi účastníky testování. Běh služby znázorněný na diagramu aktivit vidíme na obrázku 3.4. Jednotlivé aktivity jsou vysvětleny v následujících sekcích.



Obrázek 3.3: Sekvenční diagram ukázky komunikace mezi účastníky testování



Obrázek 3.4: Diagram aktivit testovací služby

3.3.1 Inicializace služby

Testovací služba začíná běh inicializační fází. Služba v inicializační fázi vytvoří připojení se všemi testovanými zařízeními dle nadefinované komunikace. Z konfigurace bude služba vědět, kolik testovaných zařízení má očekávat. Inicializační fáze zároveň bude mít definovaný časový limit, který ve výchozí konfiguraci bude 60 sekund. Zároveň ale bude nastavitelný uživatelem. Testovací služba vyhodnotí inicializační fázi jako úspěšnou, pokud se úspěšně připojí definovaný počet testovaných zařízení. Služba vyhodnocuje neúspěch inicializační fáze v případě, že nastanou tyto situace:

- vypršení časového limitu na inicializační fázi,
- chyba v komunikaci, nesprávná komunikace,
- nepřipojení definovaného počtu zařízení,
- připojení testovacího partnera.

V případě neúspěchu inicializační fáze služba nastavuje stav služby jako chybný. Testy poté nebudou provedeny.

3.3.2 Spravování testovacích partnerů

Služba bude podporovat připojení testovacích partnerů. Tato zařízení se budou připojovat před spuštěním jednotlivých testů. Testovací služba obdrží direktivu k očekávaní připojení testovacího partnera. Partner následně projde stejnou inicializační fází jako ostatní testovaná zařízení. Služba vyhodnocuje neúspěch připojení testovacího partnera v těchto případech:

- připojení jiného zařízení, než testovacího partnera,
- vypršení časového limitu na připojení,
- chyba v komunikaci.

V opačném případě služba vyhodnocuje úspěch. Po dokončení testu služba odesílá všem připojeným partnerům direktivu k ukončení testování a ukončuje spojení s nimi.

3.3.3 Registrování testů

Testovací služba nebude mít žádné informace o jednotlivých testech. Jedinou informaci, kterou testovací služba obdrží, bude číselný identifikátor testu. Reprezentace testů za pomocí čísel může způsobit kolizi mezi testovými identifikátory. Služba samotná tuto kolizi kontrolovat nebude. Způsob detekce kolize je popsán v sekci 3.7.2.

3.3.4 Testovací běh

Po úspěšné inicializační fázi a případném připojení testovacích partnerů služba čeká na direktivu ke spuštění testu. Po obdržení této direktivy s identifikátorem testu služba odesílá všem účastníkům testování zprávu k zahájení první fáze testu. Tuto fázi budeme označovat jako přípravu na testování. V této fázi účastníci testovaní připraví všechny potřebné prostředky pro provedení testu. Účastníci následně odesílají zprávu o úspěchu/neúspěchu této fáze. Služba vyhodnocuje fázi jako úspěšnou, pokud od všech účastníků obdrží zprávu o úspěchu. V opačném případě vyhodnocuje fázi jako neúspěšnou.

Služba po vyhodnocení úspěchu první fáze přechází do fáze druhé. V této fázi proběhne samotné testování. Služba odešle všem účastníkům zprávu ke spuštění této fáze. Následně očekává odpověď od všech účastníků. Služba, stejně jako v předchozí fázi, vyhodnocuje fázi jako úspěšnou, pokud obdrží zprávu o úspěchu od všech účastníků testování. V opačném případě je neúspěšná.

Poslední fází testu je úklid po testu. V této fázi služba opět vyšle zprávu k započetí fáze. Účastníci testování v této fázi uvádějí zařízení do stavu, ve kterém bylo před zahájením testu. Následně účastníci odešlou zprávu o úspěchu/neúspěchu. Vyhodnocení úspěchu/neúspěchu je stejné jako v předchozích fázích.

Po odeslání zprávy s direktivou ke spuštění jednotlivých fází testu služba čeká, než obdrží odpověď od všech účastníků testu. Tyto body, kdy služba čeká na odpověď od všech účastníků testu, budeme nazývat synchronizačními body. Účastníci, kteří dokončili svojí fázi testu, čekají na direktivu od testovací služby. Tímto se běh synchronizuje mezi všemi účastníky testu. Jednoduše lze odvodit, že během jednoho testu nastávají tři synchronizační body.

Tyto synchronizační body budou mít definovaný časový limit. Při spuštění testu bude tester moci zadat vlastní časový limit, který bude použit po každé z fází. Tedy tester bude zadávat předpokládaný maximální časový limit nejdelší fáze testování. Zároveň je ale potřeba vzít v úvahu, že časový limit je spuštěn ihned po odeslání všech zpráv. Tento časový limit bude ve výchozí konfiguraci opět 60 sekund.

Testovaná zařízení se v době běhu testu mohou dostat do chybového stavu, ve kterém nebude možné pokračovat v testování. Testovací služba tedy bude předpokládat tyto chybné stavy:

Nezajištění konzistence testování Pro konzistenci testů testovací služba vyžaduje, aby po každém testu testované zařízení bylo ve stejném stavu jako před jeho začátkem. Pokud účastník testu odešle neúspěch z důvodu vzniklé chyby ve fázi přípravy na test, nebo ve fázi úklidu po testu, tak poté bude tento účastník považován jako v chybném stavu. V případě, že chyba nastane ve fázi přípravy na test, služba neprovádí fázi testování a přechází do fáze úklidu po testu.

Vypršení časového limitu Vypršení časového limitu primárně znamená, že služba ve stanové době neobdržela odpověď od účastníka testování, ale tento účastník je stále připojen k testovací službě. Toto vede k závěru, že tento účastník je v chybovém stavu a další testování není možné.

Chyba v komunikaci Pokud během testu zařízení odpoví jinak, než dle stanovené komunikace, bude toto zařízení považováno jako by bylo v chybném stavu. Zároveň bude za chybu považováno odpojení zařízení od služby.

Pokud všechny fáze testu byly úspěšné, služba poté vyhodnocuje test jako úspěšný. Opačně pak služba vyhodnocuje test jako neúspěšný, pokud jedna z fází byla neúspěšná. V případě nastání chyby, kde testované zařízení je předpokládáno jako v chybovém stavu, testovací služba nastavuje svůj stav jako chybový. Následující testy nebudou provedeny.

3.3.5 Ukončení testování

Testovací služba po obdržení direktivy k ukončení služby odesílá všem připojeným účastníkům testování zprávu k ukončení testovacího běhu. I když služba může některého z účastníků považovat jako by byl v chybném stavu, pokud je tento účastník stále ke službě připojen, služba mu odešle tuto ukončovací zprávu. Toto se děje s cílem úspěšně ukončit co nejvíce zařízení.

3.4 Rozhraní pro testování

Důležitou součástí knihovny budou rozhraní, která umožní implementaci propojení s testovací službou a vytváření jednotlivých testů. Knihovna bude obsahovat dvě důležitá rozhraní.

3.4.1 Rozhraní testu

Jak jsem již popsal v sekci 3.3.4, jednotlivé testy budou mít tři fáze testování:

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.

Testovací knihovna tedy bude definovat rozhraní, které bude vyžadovat, aby tyto tři fáze testu byly pro každý test definované. Návrh rozhraní můžeme vidět na výpisu 3.1, kde každé fázi odpovídá jedna metoda. Tyto metody následně vrací jednoduchou informaci o úspěchu/neúspěchu. To znamená, že

vyhodnocení jednotlivých fází bude na jednotlivých testerech a jejich implementaci. Jednotlivé komponenty knihovny pouze obdrží informaci o výsledku fáze.

```
ITestCase
{
    bool StartUp()

    bool Test()

    bool TearDown()
}
```

Výpis 3.1: Návrh rozhraní pro jeden test

3.4.2 Rozhraní pro testované zařízení

Dalším důležitým rozhraním bude rozhraní pro testované zařízení. Toto rozhraní bude definovat metody, které bude potřeba definovat na každém testovaném zařízení. Cílem je, aby toto rozhraní bylo co nejjednodušší pro co nejrychlejší implementaci na nově testovaném zařízení.

Návrh tohoto rozhraní můžeme vidět na výpisu 3.2. Jak si můžeme všimnout, první tři metody se starají o propojení s testovací službou. Metoda `createConnection` bude implementovat vytvoření propojení s testovací službou. Metoda pouze vytvoří propojení s testovací službou na základě TCP/IP protokolu a následně vrátí informaci o úspěšnosti. Cíl připojení, tedy IP adresu a port, na kterém poběží testovací služba, metoda obdrží v argumentech.

O samotné odesílání, resp. přijímání jednotlivých zpráv se bude starat metoda `sendMessage`, resp. `rcvMessage`. Metoda `sendMessage` po svém zavolání převede objekt zprávy, obdržený v argumentu metody, na bajtové pole, které následně odešle testovací službě. Metoda `rcvMessage` symetricky zprávu od testovací služby přijme a převede ji na objekt jedné zprávy. Následně odkaz na tuto zprávu uloží do argumentu, který je výstupní. Obě tyto metody vrací informaci o úspěchu/neúspěchu.

Metoda `getTest` bude určena k získání jednotlivých instancí testů. Testy, které bude tato metoda vracet, musí být vytvořeny dle dříve nadefinovaného rozhraní pro jednotlivé testy. Metoda obdrží číselnou reprezentaci testu a na základě toho poté vrátí správnou instanci testu.

Metoda `getMacAddress` bude, jak již z názvu vyplývá, vracet MAC adresu zařízení. Metoda `print` bude sloužit k výpisu průběhu běhu. Tento výpis by měl hlavně posloužit k náhledu na průběh testovacího běhu, pokud se během něj vyskytnou například nějaké chyby. Nakonec metoda `stop` slouží k ukončení běhu zařízení.

```
ITestClient
{
    bool createConnection(ipAddress, port)

    bool sendMessage(message)

    bool rcvMessage(message)

    ITestCase getTest(test)

    void stop()

    void print(toPrint)

    bool getMacAddress(macAddr)
}
```

Výpis 3.2: Návrh rozhraní pro testované zařízení

3.5 Běh testovaného zařízení

O samotný běh jednotlivého testovaného zařízení se bude starat samostatná komponenta, která bude nezávislá na jakémkoliv zařízení. Tato komponenta se bude nazývat v knihovně tzv. `TestRunner`.

Komponenta bude obsahovat veškerou logiku testovacího běhu pro jednotlivé testované zařízení. Při zahájení testování komponenta za pomocí rozhraní vytvoří propojení s testovací službou a odešle zprávu o úspěchu, kde v datech zprávy uvede MAC adresu testovaného zařízení. Po obdržení zprávy o úspěchu od testovací služby komponenta přechází do metody, která bude obsluhovat příchozí zprávy.

Komponenta bude očekávat dva typy zpráv - direktivu ke spuštění testu a direktivu k ukončení testování. Po obdržení direktivy k započetí testu, komponenta ze zprávy zjistí identifikátor testu a testovací fázi. Následně z rozhraní získá instanci testu, který si po dobu běhu testu drží. Poté dle instrukcí spouští jednotlivé fáze testování. Po konci každé testovací fáze komponenta odesílá zprávu o úspěchu/neúspěchu testovací službě.

Dle této implementace může teoreticky dojít k tomu, že testované zařízení obdrží zprávu s požadavkem na spuštění fáze testování nebo fáze úklidu po testu ještě předtím, než obdrží instrukci ke spuštění fáze přípravy na test. Komponenta tuto situaci kontroluje a v případě jejího vzniku automaticky odesílá zprávu o neúspěchu fáze.

Průběh testovacího běhu testovaného zařízení lze vidět na diagramu aktivit na obrázku 3.5. Jak lze z diagramu vyčíst, běh zařízení se silně odvíjí od příkazů, které obdrží od testovací služby. Zároveň na diagramu můžeme vi-

dět již zmíněné tři synchronizační body. Tyto body jsou zvýrazněny modrým čárkovaným obdélníkem. Z diagramu se může zdát, že pokud zařízení ve fázi přípravy na testování nebo ve fázi úklidu po testu vrátí neúspěch, lze poté přesto pokračovat v testování. Toto ale není pravda, protože tento případ testovací služba nepovolí a po dokončení testu, který způsobil danou chybu, bude testovací běh ukončen. Výjimkou jsou pouze testovací partneři. Tato zařízení jsou po konci testu ukončována, proto jejich chybný stav nemá následky na další testy.

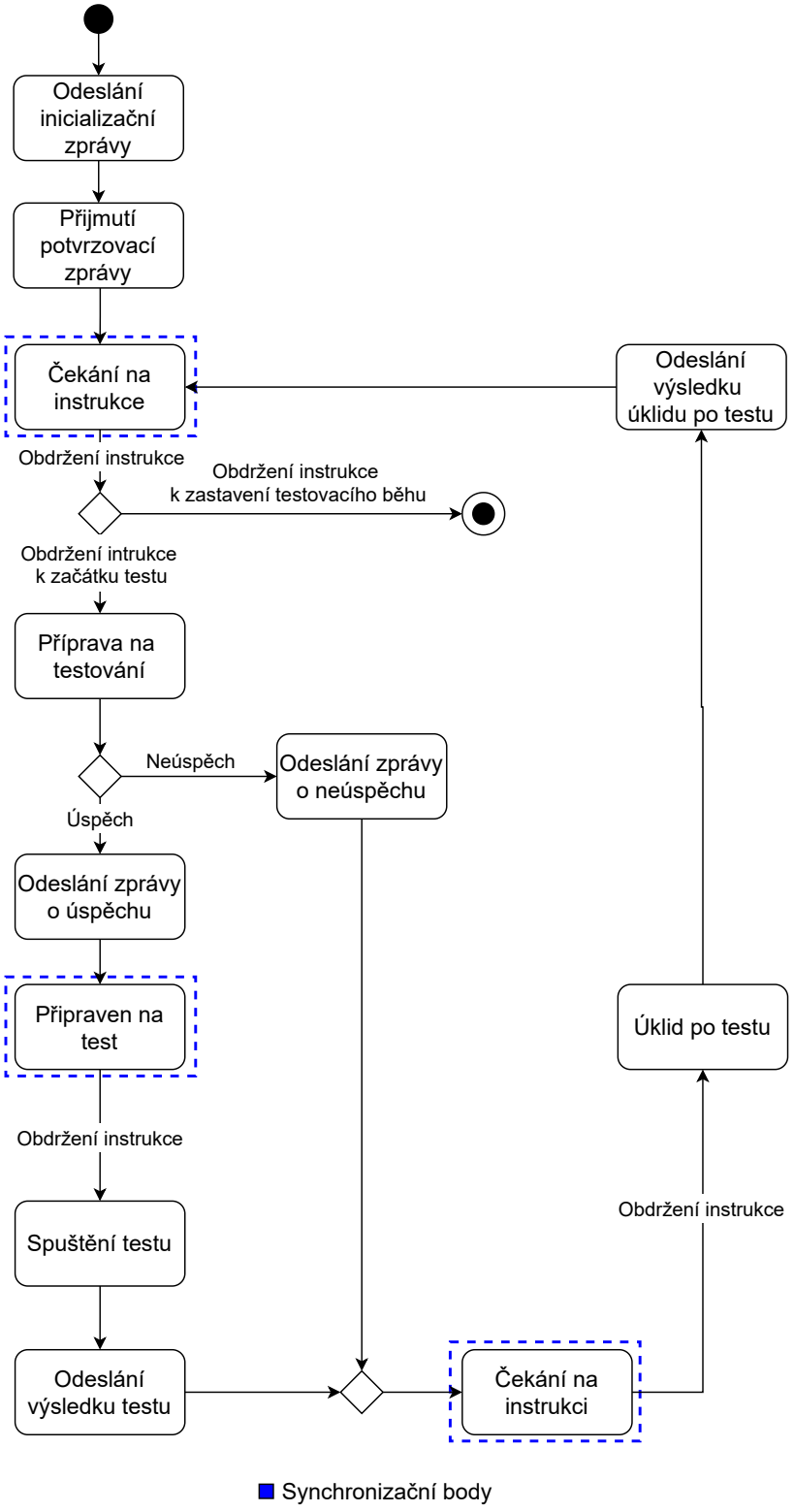
3.6 Testovací partner

Testovací partner je speciálním druhem účastníka. Toto zařízení bude spouštěno ze stejného zařízení jako testovací služba. Primárně bude fungovat na stejném principu jako testovaná zařízení. Zařízení tedy bude mít implementované rozhraní pro testované zařízení, které bude ovládáno komponentou `TestRunner`. Testovací partner ale bude také podporovat použití jakéhokoliv rozhraní pro testované zařízení.

Změnou je způsob ovládání této komponenty. Komponenta bude zaobalena vlastní třídou, která tuto komponentu bude spouštět na vlastním vlákně. Toto bude umožňovat nezávislý běh zařízení. Rozdílem také bude způsob registrování testů. Testovací partner bude podporovat dva typy způsobů, jak registrovat testy. První způsobem bude za pomocí předání jedné instance testu. Jelikož testovací partner je vázán na jednotlivé testy, je tento přístup nejjednodušší. Dalším způsobem bude předání funkce, která bude vracet instance testů na základě číselného identifikátoru. Tento způsob je totožný jako u testovaných zařízení a v určitých případech může usnadnit správu těchto zařízení.

Testovací partner bude spouštěn z testovacího projektu za pomoci testovací knihovny. Vytvořená instance partnera bude předána testovací knihovně, která se následně postará o jeho spuštění a ukončení. Zařízení bude po konci testu odpojeno od testovací služby. Testovací knihovna bude kontrolovat úspěšnost tohoto ukončení a v případě neúspěchu standardního ukončení testovací knihovna ukončí vlákno, na kterém zařízení poběží.

Testovací služba nebude nijak omezovat maximální možný počet připojených testovacích partnerů. Jediné omezení, které teoreticky může vzniknout, bude v případě, že při vytváření vláken pro jednotlivá zařízení se vyčerpá maximální počet dostupných vláken. Toto by ale při reálném použití neměl být problém, jelikož je předpokládáno, že počet těchto zařízení využitých pro jeden test se bude pohybovat maximálně v desítkách.



Obrázek 3.5: Diagram aktivit testovaného zařízení

3.7 Propojení se serverem Azure DevOps

Jedním z cílů této práce je vytvoření takového připojení, aby server Azure DevOps byl schopen spouštět jednotlivé testy a zároveň obdržel výsledek testu. K tomuto propojení využijeme testovací framework MSTest, který je nativně podporovaný serverem Azure DevOps. Celé následující propojení je možné díky uložení projektu v Azure Repos.

3.7.1 Propojení s frameworkem MSTest

Cílem vytvořeného propojení je, aby bylo co nejvíce automatizované a obsahovalo co nejméně práce ze strany testera. MSTest bude nezávisle na jednotlivých testech inicializovat službu před započnutím testování. V této době proběhne inicializační fáze služby.

Následně framework MSTest bude spouštět jednotlivé vybrané testy. V testech bude zadávat testovací službě direktivu ke spuštění testu. Zároveň bude moci před zavoláním direktivy přidávat jednotlivé testovací partnery. Následně po skončení testu framework MSTest vyhodnotí úspěšnost testu.

Po proběhnutí jednotlivých testů framework MSTest službu ukončuje. Pokud vzniknou chyby během testování, díky kterým nebude možné pokračovat v testování, bude framework vyhodnocovat následující testy jako bezvýsledné a testy přeskočí.

Propojení testovací služby a testovacího frameworku MSTest bude zajišťovat API. Toto API bude určovat přístupné metody, které mohou být využity v testovacím frameworku. Zároveň bude jedinou komponentou, kde bude framework MSTest v knihovně využívat.

3.7.2 Registrování testů

Registrování testů ve frameworku MSTest a testovací službě bude na sobě nezávislé. Framework MSTest automaticky objevuje metody, které jsou označeny jako testovací. Framework MSTest v implementaci pro jazyk C# využívá pro rozlišení testovacích součástí tzv. atributy. Atributy přidávají metadata ke třídám, typům, metodám atd. [22]. Testovací knihovna bude rozšiřovat tyto atributy o atribut `TestEnum`. Tento atribut bude identifikovat typy enumerátorů, které reprezentují testy.

Tím, že testovací služba může přijmout jakýkoliv enumerátor, může vzniknout kolize mezi jednotlivými typy enumerátorů. V inicializační fázi proto bude API kontrolovat, zda enumerátory s atributem `TestEnum` mezi sebou neobsahují kolizi a v případě jejího vzniku vyhodí výjimku a nepokračuje v testování.

Jednotlivé testy frameworku MSTest následně budou registrovány na serveru Azure DevOps. Na tomto serveru tester vytvoří tzv. Test Case, který následně asociuje odpovídající testovací metodě frameworku MSTest. Vytvoření

této asociace je nativně podporované v IDE Visual Studio 2019 [23], v němž bude celá knihovna vyvíjena.

3.7.3 Testování

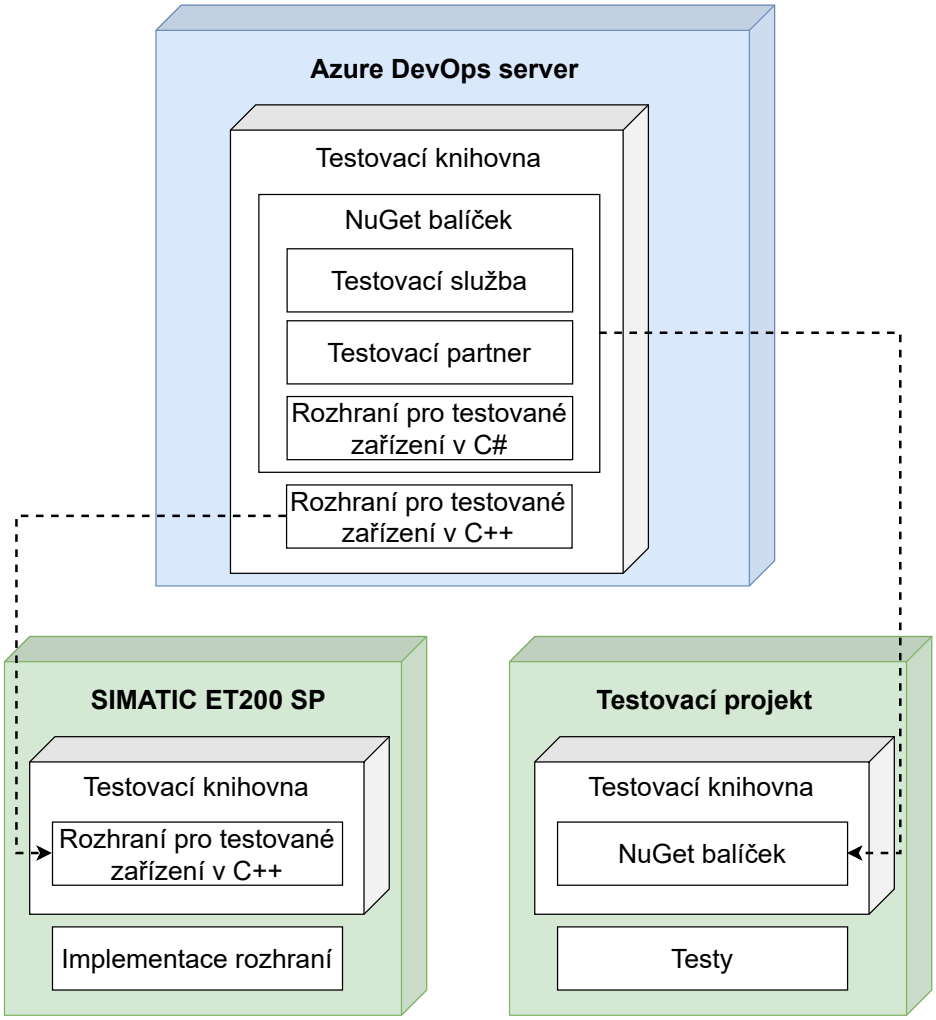
Spuštění testů na serveru je možné díky propojení Azure Pipelines a Azure Test Plans. Azure Pipelines stáhne testovací projekt na agenta z Azure Repos a následně ho dle direktiv zkompiluje. Server následně na agentovi spouští testovací projekt a spouští testy definované v určeném testovacím plánu. Tyto plány, vytvořené v Azure Test Plans, shlukují jednotlivé registrované testy a určují, které testy mají být spuštěny. Po dokončení testů server Azure DevOps automaticky obdrží výsledky testů. Tyto výsledky jsou následně dohledatelné u jednotlivých testů v Azure Test Plans a u jednotlivých běhů v Azure Pipelines.

3.8 Distribuce knihovny

Knihovna bude distribuována jako NuGet balíček. Zapouzdření knihovny do NuGet balíčku umožní jednoduché verzování, distribuci a instalaci. Zároveň NuGet balíček vytváří asociace k ostatním NuGet balíčkům, které budou použity v knihovně.

Tento NuGet balíček bude následně nahrán do služby Azure Artifacts, jejímž využitím udržíme stejnou jednoduchost distribuce NuGet balíčku a zároveň limitujeme veřejný přístup k tomuto balíčku.

Na obrázku 3.6 můžeme vidět distribuci jednotlivých částí knihovny, kam jsou její jednotlivé části distribuovány a jak je rozdělena. Součástí dříve zmiňovaného NuGet balíčku bude celá implementace v jazyku C#, což zahrnuje testovací službu, testovacího partnera a rozhraní pro testovaná zařízení v jazyku C#.



Obrázek 3.6: Model distribuce jednotlivých částí knihovny

3.9 Využití průmyslových protokolů

Jak jsem již definoval v sekci 2.5, protokoly EtherNet/IP a ModbusTCP mají jasně stanovenou komunikaci. Jejich implementace však není triviální. Z tohoto důvodu je tedy vhodné využít již vytvořené open-source knihovny, které implementují tyto protokoly. V následujících sekcích ukáži vhodné knihovny, které mohou být využity v testovací knihovně.

3.9.1 Průzkum open-source knihoven

K tomu abych mohl správně ohodnotit dostupné open-source knihovny je potřeba definovat kritéria, která musí knihovna splňovat. Jednotlivé knihovny musí být kompatibilní s vytvořenou testovací knihovnou. Tedy, knihovny by měly být implementovány v jazyce C# a kompatibilní s .NET Framework 4.8.

Vybrané knihovny budu hodnotit na základě několika faktorů. Mezi tyto faktory bude patřit:

- Aktivita autora/autorů, hodnocená na stupnici 1–10, kde 1 značí nízkou aktivitu autorů a 10 vysokou aktivitu.
- Náročnost využití knihovny, tedy náročnost implementace do knihovny a náročnost následného použití knihovny, hodnocené na stupnici 1–10, kde 1 značí nejjednodušší a 10 nejtěžší.
- Hodnocení dokumentace na stupnici 0–10, kde 0 značí, že knihovna neobsahuje žádnou dokumentaci a 10 značí nejkvalitnější dokumentaci.
- Licenční restrikce a možnosti využití knihovny v komerčním prostředí.

Následně na základě těchto kritérií a mém osobním názoru doporučím knihovnu vhodnou k použití s testovací knihovnou.

3.9.1.1 ModbusTCP

V tabulce 3.1 můžeme vidět seznam nejvhodnějších ModbusTCP knihoven pro využití v knihovně. Dle jednotlivých hodnocení je vidět, že knihovny se pohybují okolo stejné náročnosti na využití v knihovně. Všechny jsou dostupné jako NuGet balíčky, což jejich implementaci velice usnadňuje. Všechny knihovny zároveň obsahují srovnatelnou sadu funkcí.

Nejpoužívanější knihovnou, dle počtu stažení NuGet balíčku, je knihovna NModbus. Má průměrnou dokumentaci, která popisuje jednotlivé třídy. Tato knihovna je udržována komunitně, a již dvakrát byla opuštěna předchozími autory. V kontextu toho, že knihovna má nejstarší datum poslední aktualizace a žádnou viditelnou aktivitu, zde vzniká otázka, zda bude v budoucnu udržována.

Knihovna Modbus je opět komunitním projektem. I když jsou autoři vysoce aktivní, knihovna bohužel neobsahuje, až na krátkou ukázku použití, žádnou dokumentaci. Proto má knihovna v hodnocení zvýšenou náročnost na využití díky této chybějící dokumentaci.

Knihovna FluentModbus je vytvářena jednotlivcem. Knihovna obsahuje dle mého hodnocení nejlepší dokumentaci, avšak jako jediná je vydávána pod LPGL licencí, oproti ostatním knihovnám, které jsou vydávány pod licencí MIT.

Jako nejvhodnější knihovnu jsem vyhodnotil EasyModbusTCP. Knihovna je jako jediná vytvořena komerčním subjektem, a to firmou Rossmann Engineering. Z tohoto důvodu předpokládám, že na knihovnu jsou vyvíjeny vyšší kvalitativní nároky, než na ostatní knihovny.

Vybraná knihovna splňuje všechny kvalitativní kategorie a zároveň je vydávána pod MIT licencí, která je méně omezující než licence LPGL.

3.9.1.2 EtherNet/IP

Dostupných knihoven pro protokol EtherNet/IP je méně, než pro protokol ModbusTCP. Seznam dostupných knihoven můžeme vidět v tabulce 3.2. Výhodou opět je, že všechny knihovny jsou koncipovány jako NuGet balíčky. Knihovny EthernetIP a Incore bohužel neobsahují žádnou dokumentaci a obě obdržely poslední aktualizaci v roce 2018. Náročnost využití je tedy kvůli chybějící dokumentaci výrazně zvýšena. Knihovna EthernetIP také neobsahuje žádné licenční informace, oproti tomu knihovna Incore je vydána pod licencí MIT.

Nejlépe opět vyšla knihovna od firmy Rossmann Engineering, neboť jako jediná obsahuje kvalitní dokumentaci s ukázkami použití knihovny. Zároveň je vydána pod licencí MIT, díky které ji lze použít v komerčním prostředí. Proto jsem tuto knihovnu vyhodnotil jako nejvhodnější pro použití v testovací knihovně.

Název	Autor	Hodnocení aktivity	Náročnost využití	Hodnocení dokumentace	Dostupné na adrese
EasyModbusTCP	Rossmann Engineering	6	1	5	www.easymodbustcp.net
FluentModbus	Apollo3zehn	7	1	6	www.github.com/Apollo3zehn/FluentModbus
NModbus	Rich Quackenbush, et al.	4	2	4	www.github.com/NModbus/NModbus
Modbus	Andres Müller, et al.	8	3	1	www.github.com/AndreasAmMueller/Modbus

Tabulka 3.1: Seznam dostupných knihoven pro protokol ModbusTCP

Název	Autor	Hodnocení aktivity	Náročnost využití	Hodnocení dokumentace	Dostupné na adrese
EEIP	Rossmann Engineering	5	1	5	www.eeip-library.de/
EthernetIP	SecondShiftEngineer	2	4	0	www.nuget.org/packages/EthernetIP/
Incore	Yanjun Wang	2	4	0	www.github.com/wangyanjun/incore

Tabulka 3.2: Seznam dostupných knihoven pro protokol EtherNet/IP

Kapitola 4

Implementace

V této kapitole se věnuji implementaci všech navržených komponent, které umožňují automatizaci testování.

4.1 Zpráva

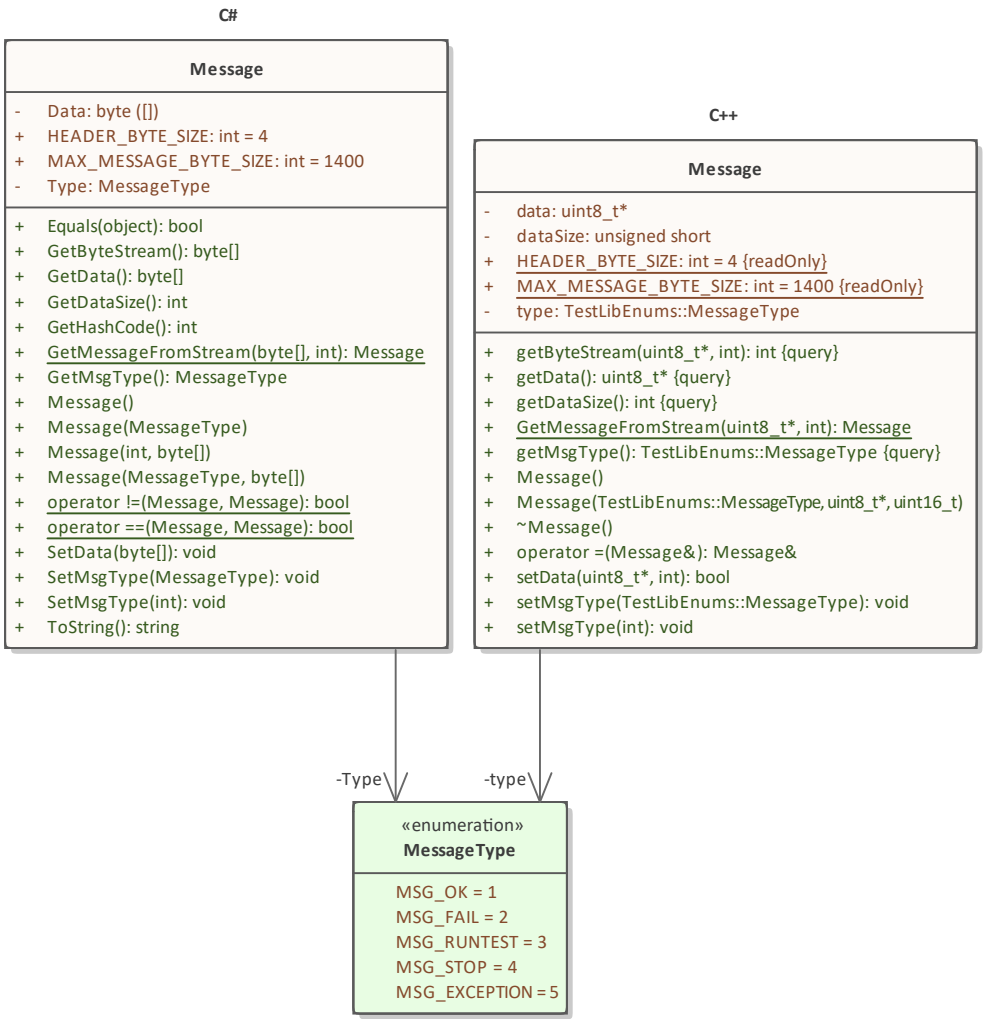
Strukturu jedné zprávy zajišťuje třída `Message`. Diagram této třídy můžeme vidět na obrázku 4.1. Třída `Message` je implementována v jazyce C# a C++.

Dle návrhu instance třídy obsahuje typ zprávy a data zprávy, pokud zpráva nějaké data obsahuje. Typ zprávy je určen enumerátorem `MessageType`. Tento výčtový typ obsahuje jednotlivé typy zpráv, definované v sekci 3.2. Zároveň tyto typy rozšiřuje o hodnotu `MSG_EXCEPTION`, která značí neplatnou zprávu.

Při odesílání a přijímání zpráv skrze protokol TCP/IP je podstatné, aby se instance třídy dala převádět na bajtové pole a obráceně. Třída `Message` proto obsahuje dvě metody, které tyto převody zajišťují. Jsou to:

- `GetByteStream` – Metoda převádí instanci třídy `Message` na bajtové pole.
- `GetMessageFromStream` – Statická metoda, která v argumentu obdrží bajtové pole a vrátí instanci třídy `Message`.

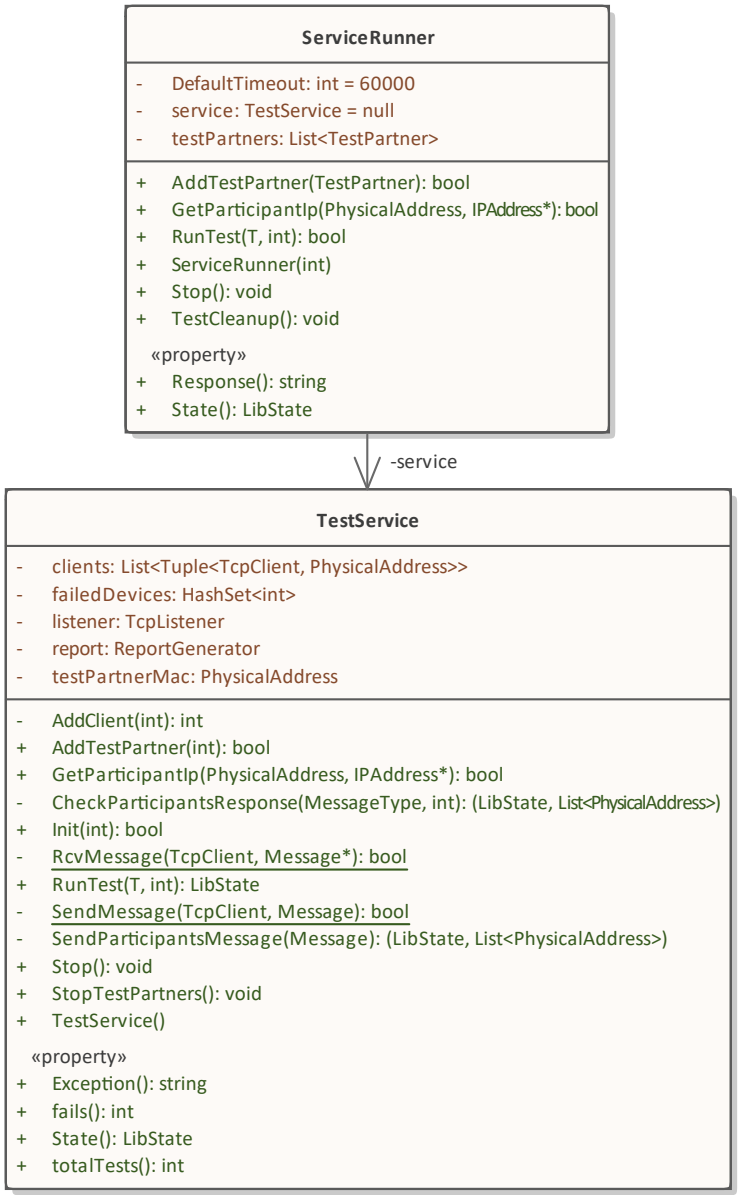
Následná komunikace skrze protokol TCP/IP je implementována v závislosti na zařízení za pomocí těchto metod. Jednotlivé implementace třídy `Message` mají mezi sebou minimální rozdíly, které jsou primárně kvůli odlišnostem jazyků C# a C++.



Obrázek 4.1: Diagram třídy Message

4.2 Testovací služba

Testovací služba je v implementaci rozdělena na dvě hlavní třídy. Třída `TestService` obstarává jednotlivé úkony služby, následně třída `ServiceRunner` propojuje třídu `TestService` s ostatními komponentami, které jsou potřebné pro testování. Tyto třídy můžeme vidět znázorněné na obrázku 4.2.



Obrázek 4.2: Diagram tříd zajišťující testovací službu

4.2.1 Nastavení služby

Konfigurace služby je uložena v souboru `config.xml`. Služba očekává tento soubor v kořenové složce, ze které je spouštěna. Tento soubor obsahuje tři hodnoty pro nastavení:

- `ip` – Adresa, na které bude služba poslouchat příchozí připojení. Všechna zařízení se budou připojovat na tuto adresu.
- `port` – Síťový port, skrz který služba provádí komunikaci.
- `participants` – Počet testovaných zařízení, která se připojí ke službě.

Ukázku tohoto souboru můžeme vidět na výpisu 4.1. V knihovně se o konfiguraci stará třída `Configuration`. Tato třída existuje v jmenném prostoru `GlobalVar`. Tento jmenný prostor simuluje globální proměnnou. Z tohoto důvodu jsou všechny třídy, které jsou odsud používány, konstruovány tak, aby primárně fungovaly jen ke čtení. Zároveň třída `GlobalVar` povolí pouze jedno přiřazení instance. Při pokusu o přiřazení jiné instance třída vyhazuje výjimku. Diagram těchto tříd můžeme vidět na obrázku 4.3. Třída rovněž obsahuje vlastnost `isConfigLoaded`, která obsahuje informaci o tom, zdali je konfigurace inicializována.

Při vytvoření instance získá třída konfiguraci z konfiguračního souboru a uloží ji do vnitřních proměnných třídy. Tyto proměnné jsou určeny pouze ke čtení, nelze je upravovat. Stav konfigurace určuje proměnná `bool IsValid`. V případě chyby je tato proměnná nastavena na hodnotu `false` a v proměnné `Exception` je uložen důvod neúspěchu. V opačném případě je tato proměnná nastavena na hodnotu `true`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
  <!-- IP of the service-->
  <ip>192.168.4.100</ip>
  <!-- Port of the service -->
  <port>1337</port>
  <!-- Number of non-virtualized participants-->
  <participants>1</participants>
</configuration>
```

Výpis 4.1: Ukázka konfiguračního souboru

4.2.2 Operace služby

Jak jsem již zmínil, jednotlivé úkony služby jsou implementovány ve třídě `TestService`. Třída po své konstrukci inicializuje vnitřní proměnné, ale ne-

provádí žádné úkony. V následujících sekcích přiblížím jednotlivé dostupné operace.

4.2.2.1 Odesílání a přijímání zpráv

Třída `TestService` vytváří během svého běhu propojení s účastníky testování. Po úspěšném připojení služba ke komunikaci používá vestavěného TCP klienta. K této komunikaci má vytvořené dvě statické metody. Jedná se o:

- `static void SendMessage(TcpClient client, Message msg)`
  Metoda pro odeslání jedné zprávy jednomu klientovi.
- `static bool RcvMessage(TcpClient client, out Message msg)`
  Metoda pro přijetí zprávy od jednoho klienta.

Všechny zprávy jsou následně odesílány a přijímány za pomoci těchto dvou metod. Rozšířením těchto metod jsou metody:

- `(LibState, List<PhysicalAddress>) CheckParticipantsResponse (MessageType expectedResponse, int timeout)`
  Metoda přijme od všech účastníků testování jednu zprávu a zkontroluje ji s očekávanou zprávou předanou v argumentu. Zároveň kontroluje, zdali obdrží zprávu v maximálním čase, který je definovaný argumentem `timeout`.
- `(LibState, List<PhysicalAddress>) SendParticipantsMessage (Message msg)`
  Metoda odešle všem participantům jednu zprávu.

Obě zmíněné metody následně v návratové hodnotě vrací dvě položky. První položkou je výsledek operace. Tento výsledek je reprezentován enumerátorem `LibState`, jenž má tyto výčty:

- `STATE_OK` - označuje úspěch,
- `STATE_FAIL` - označuje neúspěch,
- `STATE_ERROR` - označuje fatální neúspěch, který vznikl z důvodu nějaké chyby.

Tento enumerátor je zároveň hojně využíván skrze knihovnu k reprezentaci stavu různých komponent. Druhou položkou, kterou metoda vrací, je seznam zařízení, která při komunikaci způsobila neúspěch. Tento neúspěch může vzniknout ať už obdržením jiné zprávy než očekávané v případě metody `CheckParticipantsResponse`, a nebo vznikem nějaké chyby při komunikaci.

4.2.2.2 Inicializace

Třída `TestService` pro účely inicializace a přidání klienta využívá tyto dvě metody:

- `bool Init(int InitTimeout)`
  Metoda, která provádí inicializační fázi testovací služby.
- `int AddClient(int timeout)`
  Metoda pro přidání klienta do testovací služby.

Metodou `Init` třída inicializuje běh služby. Metoda z konfigurace zjistí IP adresu a port, na kterém má služba běžet. Následně začne na této IP adrese a portu poslouchat příchozí komunikaci. Z nastavení služba ví, kolik připojení má očekávat.

Proměnná `InitTimeout` určuje dobu, kdy služba čeká na příchozí komunikaci. Doba je metodě předávána, stejně jako všem ostatním metodám, které mají definovaný časový limit, v milisekundách. Metoda synchronně kontroluje, zdali nějaký účastník nečeká na připojení a zdali nevypršela maximální doba na připojení. V případě příchozí komunikace metoda zavolá metodu `AddClient`. Pokud se chce v jeden moment připojit více účastníků, jsou ostatní účastníci zařazeni do fronty.

Po zavolání metody `AddClient` metoda vytvoří připojení s testovacím zařízením a následně čeká na identifikační zprávu od testovacího zařízení, nejdéle však po dobu definovanou argumentem `timeout`.

Po obdržení zprávy metoda uloží vytvořeného klienta a jeho MAC adresu do seznamu připojených účastníků testování. Nakonec metoda vrací tři hodnoty typu integer:

- `0` – pokud připojení účastníka testování vyústilo v neúspěch, ať už kvůli překročení časového limitu, nebo kvůli nedodržení stanovené komunikace,
- `1` – pokud se ke službě úspěšně připojí testované zařízení,
- `2` – pokud se ke službě úspěšně připojí testovací partner.

Metoda `AddClient` považuje za testovacího partnera takového účastníka, který odešle jako svoji MAC adresu adresu testovacího partnera. Zařízení, která odešlou jinou MAC adresu, jsou považována za testovaná zařízení. Metoda považuje nulovou MAC adresu jako neplatnou.

Služba úspěšně ukončuje inicializační fázi poté, co se úspěšně připojí stejný počet účastníků testování, jak bylo určeno v konfiguraci. V případě nepřipojení se očekávaného počtu zařízení v definovaném čase, obdržení špatné nebo žádné zprávy služba vyhodnocuje inicializační fázi jako neúspěšnou. Připojení testovacího partnera v této fázi vyústí též v neúspěch inicializace.

4.2.2.3 Připojení testovacího partnera

Jelikož každý test může obsahovat různý počet testovacích partnerů, je podstatné, aby služba mohla při testovacím běhu tato zařízení přidávat a odebírat. K tomu slouží dvě metody. Metoda `AddTestPartner` řekne službě, že má očekávat připojení testovacího partnera. V případě úspěšného připojení metoda vrací úspěch. Naopak v případě chyby, nebo připojení jiného zařízení, metoda vrací neúspěch. O opačnou operaci se stará metoda `StopTestPartners`. Metoda po svém zavolání odešle všem testovacím partnerům zprávu o ukončení testování a tato zařízení jsou odpojena a ukončena.

4.2.2.4 Spuštění testu

Nejpodstatnější operací je spuštění jednotlivých testů. O to se stará metoda `LibState RunTest<T>(T testEnum, int timeout)`, která očekává dva parametry:

- `T testEnum` – Identifikátor testu s generickým typem T, který typem musí být enumerátor.
- `int timeout` – Maximální délka, po kterou služba očekává odpověď od účastníků testu.

Metoda odesílá všem účastníkům testování direktivu k započnutí jednotlivých fází testu. Poté čeká na odpověď od všech účastníků testu. Doba čekání je určena argumentem `timeout`. Tento čas je vázaný na jednotlivá stádia testování. Tedy pokud má metoda časový limit 30 sekund, bude po každém odeslání direktivy k započnutí fáze testu čekat na odpověď maximálně těchto stanovených 30 sekund. Tento časový limit je realizován za pomoci metody `CheckParticipantsResponse`.

Metoda během běhu vyhodnocuje, zda se účastník testování nedostal do chybového stavu. Na základě toho poté spouští jednotlivé fáze. Po skončení testu metoda vrací jednu z hodnot enumerátoru `LibState`.

4.2.2.5 Ukončení testování

Po dokončení celého testovacího běhu je zavolána metoda `Stop()`. Ta odešle všem stále připojeným účastníkům direktivu k ukončení testování a ukončí spojení. Služba se snaží v případě chyby ukončit co nejvíce zařízení skrze stanovený protokol.

4.2.2.6 Pomocné metody a třídy

Třída `TestService` obsahuje navíc pomocnou metodu `GetParticipantIp`. Tato metoda slouží k získání IP adresy testovaného zařízení na základě znalosti MAC adresy zařízení. Pokud je zařízení s MAC adresou, kterou metoda

obdrží v argumentu funkce, připojeno, tak poté metoda ukládá do výstupního argumentu jeho IP adresu a vrací úspěch. V opačném případě vrací neúspěch a jako IP adresu ukládá hodnotu `null`.

Třída také využívá při svém běhu dvě pomocné třídy, jejichž diagram můžeme vidět na obrázku 4.3. První třídou je třída `Logger`. Tato třída realizuje zapisování protokolu průběhu testovacího běhu do souboru. Zároveň je konstruována tak, že může zapisovat maximálně jedno vlákno programu současně.

Třída je taktéž obsažena ve třídě `GlobalVar`, která simuluje globální proměnnou. Inicializaci instance třídy `Logger` lze provést odkudkoliv, avšak stejně jako u konfigurace pouze jen jednou. Třída `GlobalVar` obsahuje vlastnost `isLogEnabled`, která obsahuje informaci o tom, zdali je zapisování inicializováno.

Třída `TestService` obsahuje ve svých metodách kontrolu, zdali je instance třídy `Logger` inicializována a v případě kladného vyhodnocení zapisuje průběh svého běhu. Instance třídy `Logger` by teda měla být vytvořena ještě před započnutím běhu služby.

Druhou třídou, kterou třída `TestService` využívá, je třída `ReportGenerator`. Tato třída vytváří výstupní XML soubor s výsledky jednotlivých testů. Tento soubor je vytvořen ve formátu dle frameworku NUNit, jehož definici můžeme najít v [24]. Tento soubor následně může být nahrán do serveru Azure DevOps. V navrženém řešení je tento výstup považován za záložní.

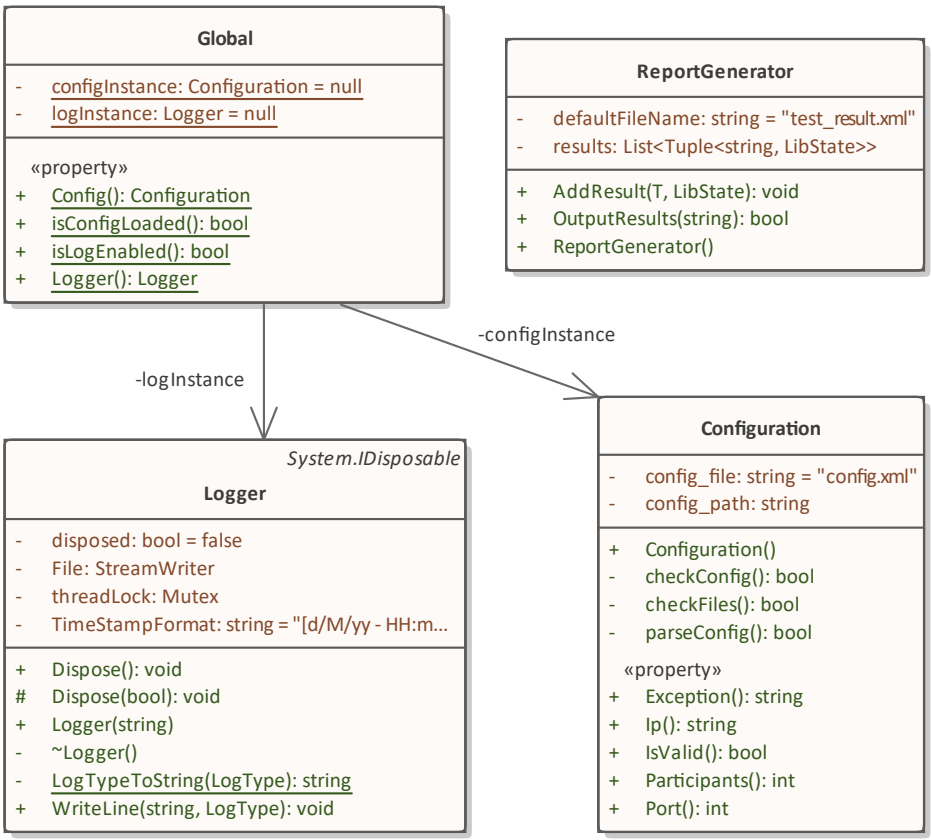
4.2.3 Propojení s ostatními komponentami

O propojení třídy `TestService` s ostatními komponentami knihovny se stará třída `ServiceRunner`. Tato třída ve svém konstruktoru inicializuje konfiguraci služby a zjistí, zda je validní. Tento konstruktor přijímá jeden argument – časový limit na inicializaci. V případě nezadání tohoto parametru třída využije defaultní časový limit definovaný ve třídě na 60 sekund. Následně konstruktor vytvoří instanci třídy `TestService` a zavolá metodu `Init`. V případě jakékoliv chyby je služba vyhodnocena jako by byla v chybovém stavu. Toto určuje enumerátor `LibState` v proměnné `State`. Třída využívá pouze hodnoty `STATE_OK` a `STATE_ERROR`. Následný důvod vyhodnocení chyby je uložen v proměnné `Response`. Třída má další metody, skrze které umožňuje ovládání instance třídy `TestService`. Jsou to tyto metody:

- `bool RunTest<T>(T test, int timeout = DefaultTimeout)`
  Metoda předá instrukci pro spuštění testu. Očekává stejné parametry jako metoda `RunTest` třídy `TestService`. Jedinou změnou je, že pokud časový limit nebude zadán, bude použit defaultní časový limit.

- `bool AddTestPartner(TestPartner participant)`
  Metoda spouští testovacího partnera předaného v argumentu a předává informaci o očekávání jeho připojení instanci třídy `TestService`. Instanci partnera následně uloží do seznamu partnerů.

- `void TestCleanup()`
  Metoda je volána po dokončení jednotlivého testu, předá instanci třídy `TestService` direktivu k odpojení testovacích partnerů. Následně zkontroluje, zda se partneři ukončili a pokud ne, tak je ukončí.
- `void Stop()`
  Metoda předá informaci o ukončení testovacího běhu.
- `bool GetParticipantIp(PhysicalAddress macAddr,out IPAddress addr)`
  Zpřístupnění stejnojmenné metody z třídy `TestService`.



Obrázek 4.3: Diagram tříd, které zajišťují pomocné služby

4.3 Rozhraní pro testovaná zařízení

Diagram tříd, které realizují implementaci pro účastníka testování, můžeme pro implementaci v jazyku C++ vidět na obrázku 4.4 a pro implementaci v jazyku C# na obrázku 4.5. Obě implementace obsahují rozhraní pro testované zařízení, které je následně na testovaném zařízení implementováno testerem.

Vytvořená rozhraní jsou následně využita ve třídě `TestRunner`. Tato třída zajišťuje běh jednotlivých účastníků testování. Obsahuje tyto metody:

- `Init(ipAddress,port)`
  Metoda inicializuje připojení s testovací službou, kde parametry připojení ke službě jsou obdrženy v argumentech funkce.
- `HandleInstructions()`
  Metoda přijímá instrukce od testovací služby a na jejich základě provádí úkony.
- `RunTest(testIdentifier, testState)`
  Metoda spouští jednotlivé fáze testů.
- `Stop()`
  Metoda ukončuje testovací běh.

Bližší vysvětlení si zaslouží metoda `RunTest`. Tato metoda je volána z metody `HandleInstructions`. K udržení stádia testu si třída mezi jednotlivými stádii drží instanci jednotlivých testů. Ve fázi přípravy na testování metoda zkontroluje, že je instance testu nastavena na hodnotu `null`. Jiná hodnota by značila chybu během předchozího testování. Ve fázi úklidu po dokončení testu je odkaz na instanci testu nastaven zpět hodnotu null. Třída `TestRunner` je opět implementována v jazyce C# a C++.

Návrhově je rozhraní pro zařízení i třída `TestRunner` v obou implementacích ekvivalentní. Je třeba ale upřesnit implementaci v C++. Tato implementace je mířena na testované zařízení, které běží na vlastním speciálně vyvinutém hardwaru. Zařízení však v některých případech nepoužívá některé vestavěné funkce a místo nich používá jejich vlastní náhradu.

Implementace v C++ tedy definuje seznam funkcí, které jsou potřeba implementovat, resp. je potřeba u nich vytvořit odkaz na funkce, které jsou implementované na zařízení. Seznam těchto funkcí můžeme vidět na výpisu 4.2. Z něj lze vyčíst, že se jedná o funkce, které alokují a uvolňují paměť na haldě, a funkci, které paměť kopíruje.

```
/*  Allocation of memory on heap */
void* TESTLIB_ALLOC_MEM(const uint32_t length);

/* Free of allocated memory */
bool TESTLIB_FREE_MEM(void* ptr);

/* Copy of memory */
void* TESTLIB_MEMCPY(void* const lpDst, const void* const lpSrc, int
↪  dwNb);
```

Výpis 4.2: Seznam funkcí k implementaci na zařízení v jazyce C++

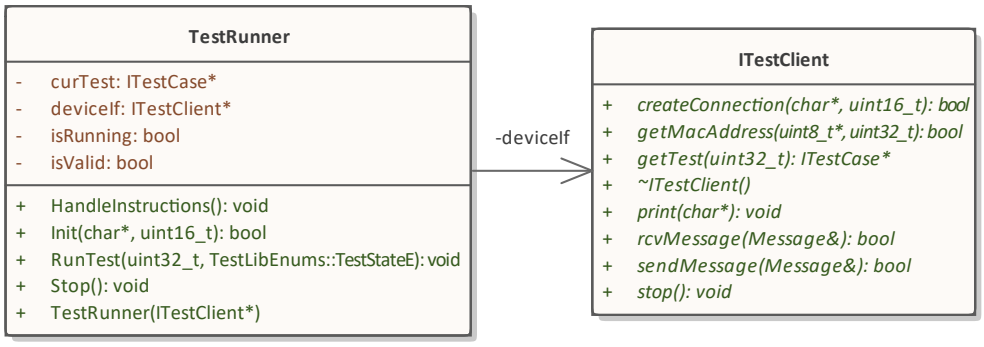
4.4 Testovací partner

Implementace testovacího partnera přímo kopíruje navržené rozhraní pro testované zařízení. Jeho logika je obsažena ve třídě `TestPartner`. Diagram třídy můžeme vidět také na obrázku 4.5. Tato třída má tři možné konstruktory:

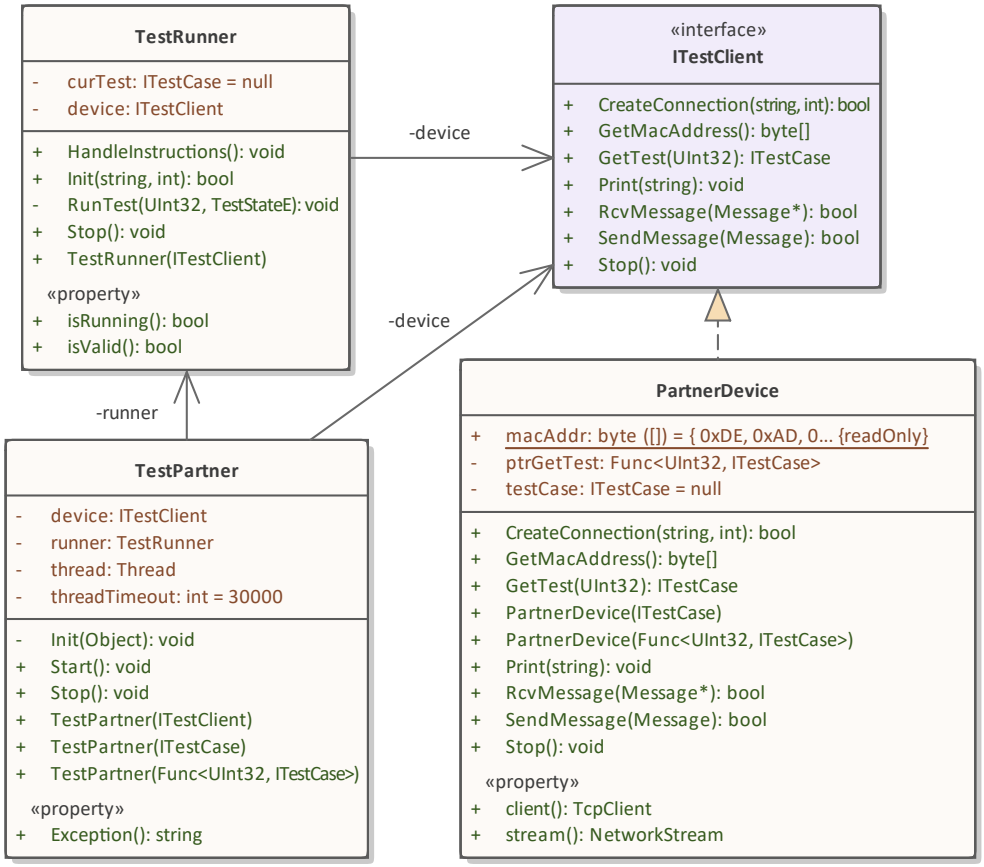
- `TestPartner(ITestClient device)`
  Konstruktor dostane v argumentu odkaz na instanci rozhraní zařízení.
- `TestPartner (ITestCase testCase)`
  Konstruktor dostane v argumentu odkaz na instanci jednoho testu, který je odvozený z rozhraní pro test.
- `TestPartner (Func<UInt32, ITestCase> getTest)`
  Argumentem konstruktoru je funkce, která obsahuje seznam testů. Tato funkce musí po obdržení číselné reprezentace testu v argumentu vrátit instanci testu, který je odvozený z rozhraní testu.

Třída následně vytvoří instanci implementovaného rozhraní pro zařízení (třída `PartnerDevice`), který přijme v konstruktoru vybraný způsob reprezentace testů. Instanci rozhraní následně předá instanci třídy `TestRunner` a po zavolání metody `Start` spustí instanci třídy `TestRunner` na vlastním vlákně.

Třída rovněž obsahuje metodu `Stop`, která po zavolání kontroluje ukončení testovacího partnera v daném časovém limitu. Po vypršení časového limitu funkce ukončuje vlákno, na kterém instance třídy `TestRunner` běží.



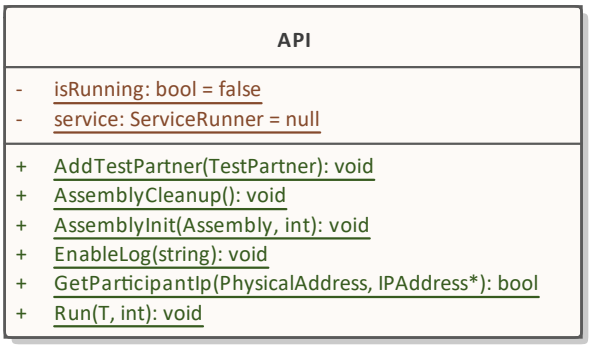
Obrázek 4.4: Diagram tříd implementace účastníka testování v jazyce C++



Obrázek 4.5: Diagram tříd implementace účastníka testování v jazyce C#

4.5 Propojení s frameworkem MSTest

Podstatnou součástí implementace je propojení služby s testovacím frameworkem MSTest. O toto se stará třída `API`, jejíž diagram můžeme vidět na obrázku 4.6. Jako jediná využívá nástroje tohoto frameworku. Je koncipována jako statická třída, i když je závislá na jím vytvořené instanci třídy `ServiceRunner`. Toto je utvořeno kvůli kompatibilitě s frameworkem MSTest.



Obrázek 4.6: Diagram třídy API

4.5.1 Použití frameworku

Jak jsem již zmínil v sekci 3.7.2, framework MSTest pro identifikaci testových metod využívá atributy, například:

- `AssemblyInitialize` – identifikuje inicializační funkci, která je spuštěna před spuštěním testů [25].
- `AssemblyCleanup` – identifikuje funkci, která bude spuštěna po skončení všech testů [25].
- `TestClass` – identifikuje třídu, která obsahuje testy [26].
- `TestMethod` – identifikuje metody, které reprezentují jednotlivé testy [26].

Framework MSTest následně používá tyto metody, které jsou identifikovány těmito a dalšími atributy. Při kompilaci jazyk C# vytváří jednotlivé celky kompilovaného kódu, které poté spojuje do logického celku. Tyto jednotlivé celky se nazývají *assembly*. [27]

Implementace testovací knihovny a uživatelské použití knihovny, a tedy i testovacího frameworku MSTest, tvoří samostatné logické celky. Framework MSTest následně detekuje správně definované funkce pouze z toho logického celku, ze kterého je spouštěn.

4.5.2 Spuštění a ukončení služby

Při spuštění jednotlivých testů je nutné brát v potaz, že testovací služba neví, jaké testy budou kdy spuštěny. Testovací služba se tedy musí používat nezávisle na všech testech. K tomuto využijeme dvě funkce frameworku MSTest – `AssemblyInitialize` a `AssemblyCleanup`. Tyto funkce jsou spuštěny před započetím a po skončení testovaní. Ve třídě `API` jejím implementacím odpovídají metody `AssemblyInit` a `AssemblyCleanup`. Metoda `AssemblyInit` přijímá dva argumenty:

- `Assembly assembly` – odkaz na runtime blok, který odkazuje na uživatelskou část použití testovací knihovny.
- `int InitTimeout` – časový limit na inicializační fázi služby (tedy časový limit, který je využíván v metodě `TestService.Init`), pokud je ponechán prázdný, je použit defaultní časový limit z třídy `ServiceRunner`.

Metoda `AssemblyInit` na počátku kontroluje kolize mezi testovými identifikátory. Způsob kontroly je popsán v sekci 4.5.3. Metoda následně inicializuje službu, která projde inicializační fází.

Metoda `AssemblyCleanup` symetricky ukončuje běh testovací služby. V určitých případech tato funkce nemusí být zavolána. Při zjištění chyby v běhu, která vyústí v ukončení testovacího běhu, je tedy tato funkce zavolána manuálně. Zároveň je kontrolováno, aby tato metoda nebyla volána vícekrát.

Aby mohl MSTest rozeznat tyto funkce, je potřeba je definovat ve stejné assembly, která spouští tento framework. Toto vytváří problém u automatizace vytvoření propojení s testovací knihovnou. Pokud v testovací knihovně definujeme metody, které budou obsahovat atributy `AssemblyInitialize` a `AssemblyCleanup`, testovací framework MSTest tyto metody nevidí, a tak je nepoužije. Proto je potřeba vytvořit dvě funkce, které budou obalovat metody `AssemblyInit` a `AssemblyCleanup`, ve stejné assembly jako ty, ve které běží framework MSTest.

Toto spravuje statická třída `TestLibInit`,která pouze zaobaluje tyto dvě metody svými metodami a přidává k nim potřebné atributy. Třída není součástí assembly testovací knihovny. V kontextu samotné knihovny je to pouze textový soubor. Proces propojení této třídy s testovacím projektem, ve kterém poběží framework MSTest, je popsán v sekci 4.6.

4.5.3 Registrování testů

Při rozlišování jednotlivých testů knihovna a testovací služba využívá enumerátory, jenž odkazují na nějakou číselnou hodnotu, která je následně odesílána účastníkům testování. Všechny enumerátory musí být označeny atributem `TestEnum`. Díky němu může knihovna identifikovat výčtové typy, které reprezentují testy a zjistit kolize mezi nimi.

Tato kontrola je provedena před započetím testování v metodě `AssemblyInit` díky odkazu na runtime blok, který obdrží v argumentu. Tato kontrola se dá přeskočit předáním hodnoty `null` jako odkazu na runtime blok v argumentu metody.

Framework MSTest registruje testy nezávisle na testovací knihovně. Jednotlivé třídy musí být označeny atributem `TestClass` a zároveň jednotlivé metody atributem `TestMethod`. Framework následně testy detekuje automaticky.

4.5.4 Spuštění testu

Pro spuštění testu je použita metoda `Run` z třídy `API`. Tato metoda přijímá stejné argumenty jako metoda `RunTest` třídy `ServiceRunner`. Metoda zkontroluje, že argument identifikující test je typem enumerátor a obsahuje atribut `TestEnum`. V případě, že se testovací služba nenachází v chybném stavu, předá metoda testovací službě direktivu k započetí testu. Následně vyhodnotí jeho úspěšnost. Je-li služba je v chybném stavu, metoda vyhodnotí test jako bezvýsledný.

Zároveň lze před započetím testu přidat testovací partnery. Toto je možné za pomocí metody `AddTestPartner` z třídy `API`. Metoda v argumentu obdrží instanci testovacího partnera a následně se postará o jeho běh.

4.5.5 Pomocné metody

Jedním z návrhových cílů třídy `API` je, aby všechny uživatelsky podporované operace šli provést skrz tuto třídu. Třída proto obsahuje tyto pomocné metody:

- `static void EnableLog(string fileName = "log.txt")`
  Metoda inicializuje zápis běhu testovací knihovny, který je realizovaný třídou `Logger`. Název výstupního souboru je předán v argumentu, případně je použit defaultní název.

- `static bool GetParticipantIp(PhysicalAddress macAddr, out IPAddress addr)`
  Zpřístupnění stejnojmenné metody z třídy `ServiceRunner`.

4.6 Vytvoření balíčku NuGeT a distribuce knihovny

Knihovna je distribuována jako NuGet balíček. Vytvořený balíček NuGet je následně nahrán do služby Azure Artifacts. K jeho vytvoření stačí definovat konfigurační soubor s příponou `.nuspec`, který definuje metadata balíčku a závislosti na ostatních NuGet balíčcích. Balíček je následně vytvořen za pomocí příkazu `nuget pack`.

NuGet balíček umožňuje i distribuci souborů, které nejsou součástí kompilované knihovny. Tuto funkci využijeme k distribuci několika souborů. Balíček obsahuje informační soubor, který popisuje základní informace k použití balíčku. Taktéž ale obsahuje soubory, které usnadňují použití knihovny.

Prvním je `TestLibInit.cs`. Tento soubor obsahuje třídu `TestLibInit`, která obaluje metody `AssemblyInit` a `AssemblyCleanup` třídy `API`. Soubor je díky NuGet balíčku automaticky vytvořen při instalaci a zároveň je automaticky přidán mezi soubory, které jsou v testovacím projektu kompilovány.

Dalším souborem je soubor `TestEnumTemplate.cs.txt`. Tento soubor obsahuje šablonu souboru, který bude obsahovat enumerátory, které identifikují testy. Tato šablona obsahuje preprocesorové direktivy, které dělají tento soubor kompatibilní jak pro jazyk C#, tak pro jazyk C++. Soubor má přidanou příponu `.txt` kvůli tomu, aby nebyl automaticky přidán do kompilace testovacího projektu.

Posledním souborem je soubor `config.xml`. Tento soubor obsahuje nastavení služby, které již bylo definováno v sekci 4.2.1. Všechny tyto soubory jsou po instalaci uloženy do složky `resources`.

Je dobré vytvořit kopii souborů `TestEnumTemplate.cs.txt` a `config.xml` a přesunout je mimo složku `resources`. Při aktualizaci NuGet balíčku jsou totiž aktualizovány všechny soubory, které jsou na něj vázané. Toto může zapříčinit přepsání konfiguračních souborů, pokud nebudou přesunuty.

Kapitola 5

Demonstrace použití knihovny

V této kapitole se zabývám demonstrací použití knihovny při testování. Ukázkové testy využívají průmyslový protokol ModbusTCP.

5.1 Nastavení knihovny

Před započetím testování je potřeba provést prvotní nastavení testovací služby a testovaného zařízení.

5.1.1 Příprava testovaného zařízení

K použití testované knihovny na testovaném zařízení je potřeba implementovat rozhraní pro testované zařízení. Instance tohoto rozhraní následně musí být předána komponentě `TestRunner`. Implementace obou těchto částí je popsána v sekci 4.3. Komponenta `TestRunner` je následně inicializována za pomocí metody `Init` a následně je zavolána metoda `HandleInstructions`, která řídí testovací běh na zařízení.

5.1.2 Testovací projekt

Ke správě jednotlivých testů je potřeba vytvořit testovací projekt, ze kterého následně bude spouštěna testovací služba. Testovací projekt je vytvořen za pomocí IDE Visual Studio 2019, kde je využito předlohy nazvané „Unit Test Project (.NET Framework)“.

Do tohoto projektu je následně přidán NuGet balíček, který obsahuje testovací knihovnu. Jelikož je knihovna uložena v Azure Artifacts, může být potřeba zároveň přidat Azure Artifacts jako zdroj NuGet balíčků. S instalací knihovny se přidá i složka `resources`, která obsahuje vše potřebné k nastavení testovací knihovny a testovací služby. Testovací projekt je následně uložen ve vlastním repositáři v Azure Repos.

5.1.2.1 Konfigurace

Ve složce `resources` nalezneme soubor `config.xml`, který obsahuje konfiguraci služby. Tento soubor po zkopírování ze složky resource do kořenové složky projektu lze upravit do požadovaného základního nastavení.

Testovací knihovna a služba očekává tento soubor v kořenové složce, ze které se spouští program. Soubor je tedy potřeba po kompilaci do této složky přesunout. Toho lze docílit za pomocí tzv. „Post-build events“, neboli událostí po sestavení. Do těchto událostí je přidána direktiva, která po kompilaci, neboli sestavení, přesune konfigurační soubor do výstupní složky s programem. Tuto direktivu můžeme vidět na výpisu 5.1.

V tento moment je zároveň vhodné vytvořit soubor, ve kterém budou uloženy jednotlivé identifikátory testů. K tomu je využit soubor `TestEnum-Template.cs.txt` ve složce `resources`. Jeho kopie je přesunuta mimo složku resources a následně je přejmenován na `TestEnum.cs`.

```
COPY "$(ProjectDir)config.xml"  "$(TargetDir)"
```

Výpis 5.1: Direktiva k přesunutí konfiguračního souboru

5.2 Vytvoření testu

Jednotlivé testy je potřeba implementovat pro všechny účastníky testování. Pro demonstraci funkcionality jsou vytvořeny dva testy:

1. `MODD_ADD_CONF` – Test, který otestuje přidání nastavení implementace protokolu ModbusTCP na testovaném zařízení.

2. `MODD_READ_DATA` – Test, při kterém se otestuje čtení z registrů na testovaném zařízení za pomoci osmi testovacích partnerů. Partneři budou číst z testovaného zařízení za pomoci protokolu ModbusTCP a následně kontrolovat správnost obdržených dat.

Ke každému z testů je vytvořen stejnojmenný identifikátor v souboru `TestEnum.cs`.

5.2.1 Testované zařízení

Implementace testu `MODD_ADD_CONF` v první fázi nic neprovádí. Následně ve druhé fázi je testována metoda `mfd_modd_add_configuration`, která vytváří konfigurace protokolu ModbusTCP. V poslední fázi jsou tyto konfigurace odstraněny.

Implementaci testu `MODD_READ_DATA` lze vidět na výpisu 5.2. Test v první fázi inicializuje protokol ModbusTCP a do výstupních registrů zapíše data. Jeden registr má velikost dva bajty. Během testovací fáze zařízení neprovádí

žádné úkony. Zařízení nakonec vrací v poslední fázi testu do původního stavu před testováním.

Vytvořené testy jsou následně přidány do metody `getTest` v rozhraní pro testované zařízení. Metoda `getTest` musí po obdržení číselného identifikátoru jednotlivých testů vrátit instanci těchto testů.

5.2.2 Testovací partner

Implementace testu `MODD_READ_DATA`, kterou můžeme vidět na výpisu 5.3 pro testovacího partnera je velmi obdobná implementaci pro testované zařízení. Implementace používá knihovnu EasyModbusTCP k vytvoření připojení s testovaným zařízením.

Testovací partner v testovací fázi vytvoří připojení s testovaným zařízením a poté čte data z testovaného zařízení, které následně kontroluje s očekávanými daty. Tato implementace je součástí testovacího projektu. Test `MODD_ADD_CONF` nevyužívá testovacího partnera.

5.2.3 Testovací služba

K registraci testů pro testovací službu je použita standardní metodika vytváření testů dle frameworku MSTest, kterou můžeme vidět na výpisu 5.4, kde lze vidět testovací třídu `Modbus` obsahující testovací metody `TestModdAddConf` a `TestModdReadData`. Třída i metody jsou označeny atributy definovanými frameworkem MSTest.

Uvnitř testovací metody `TestModdAddConf` můžeme vidět direktivu k započetí testu definovanou testovací knihovnou, která spouští test `MODD_ADD_CFG`. Tato direktiva obdrží v argumentu enumerátor reprezentující test. Tento test nevyužívá testovacích partnerů.

V testovací metodě `TestModdReadData` můžeme vidět cyklus, který přidá osm testovacích partnerů, v argumentu direktivy vytváření instance testovacího partnera, který ve svém argumentu obdrží instanci testu `TestModdReadData`. Nakonec je v metodě direktiva k započetí testu, která je předána testovací službě.

Takto vytvořené testy lze následně vidět v IDE Visual Studio 2019 v průzkumníku testů. Z tohoto průzkumníku můžeme testy spouštět a otestovat tak jejich funkčnost.

5.3 Propojení s Azure DevOps serverem

Propojení s Azure DevOps serverem funguje v závislosti na kooperaci několika služeb tohoto serveru. Jejich propojení je možné vytvořit díky tomu, že všechny části jsou uloženy v Azure Repos.

5.3.1 Příprava

K tomu aby server Azure DevOps mohl samostatně spouštět testy, je potřeba vytvořit v Azure Pipelines seznam pravidel, který zkompiluje testovací projekt, neboli tzv. „pipeline“. Tato pipeline je nastavena tak, aby na začátku stáhla z Azure Repos testovací projekt. Seznam obsahuje tato pravidla:

1. Obnovení NuGet balíčků.
2. Kompilaci testovacího projektu.
3. Nahrání artefaktů kompilace.

Bližší přiblížení si zaslouží nahrání artefaktů kompilace. Za artefakty se v tomto případě považují výstupní soubory kompilace. Tedy kompilace je provedena do pracovní složky, která je následně nahrána pro další použití.

Následně vytvořím tzv. „Release pipeline“. Ta využije artefakty vytvořené v předchozí fázi. Zároveň využije artefakty, které vznikly z kompilace zdrojového kódu testovaného zařízení. Kompilace je prováděna automaticky po nahrání jakékoliv změny kódu.

Do nově vytvořené pipeline přidám fázi, ve které spustíme testování. Fáze je nastavena tak, aby se spouštěla na správném zařízení, které obsahuje potřebné propojení s dalším hardwarem potřebným ke spuštění testovaného zařízení. Do této fáze je též potřeba přidat pravidlo, které před spuštěním testování nahraje aktuální verzi zdrojového kódu na testované zařízení a zároveň ho spustí. Rovněž lze přidat pravidlo, které upravuje nastavení testovací služby dle potřeby. Po dokončení konfigurace je spuštěno samotné testování.

Poté vytvořím testovací plán. Do nastavení testovacího plánu přidám již vytvořené pipeline a následně budou přidány i všechny testy.

5.3.2 Registrování testů

Testy vytvořené skrze framework MSTest lze registrovat v Azure Test Plans. Nejdříve je potřeba vytvořit testovací případ (neboli v angličtině tzv. „Test Case“) na serveru. Po jeho vytvoření se testu přiřadí unikátní číselný identifikátor. Test bude následně přidán do dříve vytvořeného testovacího plánu.

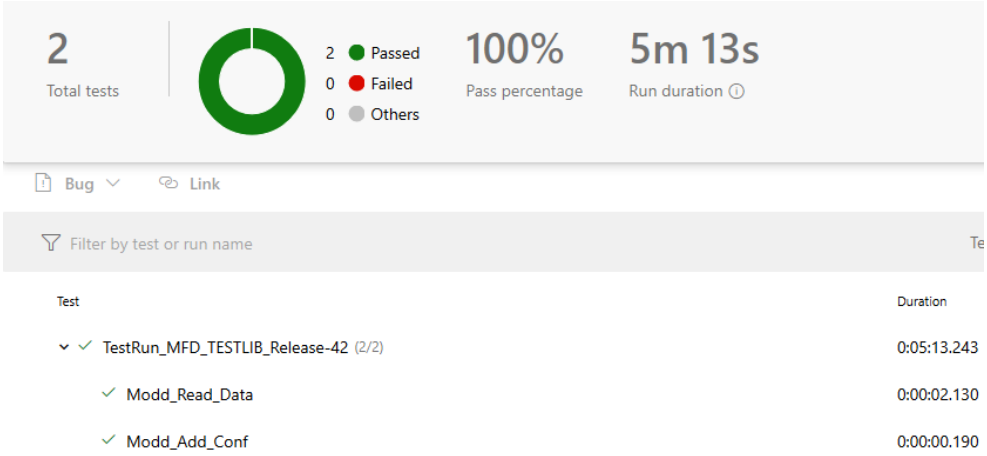
Po vytvoření testovacího případu je potřeba vytvořit asociaci mezi testovacím případem na serveru Azure DevOps a testem v testovacím projektu. To lze udělat za pomocí IDE Visual Studio 2019. Po vytvoření připojení mezi Azure DevOps serverem a IDE Visual Studio lze z průzkumníku testů vytvořit asociaci mezi testovacím případem a testovací metodou. Pravým kliknutím na test vyberu „Přidružit k testovacímu případu“ a do otevřeného okna následně vložím identifikátor testovacího případu a potvrdím přidružení. Testovací případ je následně nastaven do stavu „Automatizovaný“.

5.4 Spuštění testů

Po všech předchozích konfiguracích je následně možné spouštět registrované testy a testovací plán, který se skládá z registrovaných testů. Způsob spuštění testů je pak již pouze na nastavení. Testy lze například spustit:

- manuálně,
- automaticky při přidání změn do projektu,
- automaticky dle časového plánu.

Server Azure DevOps následně obdrží výsledky o testování. Jeden z možných způsobů zobrazení výsledků můžeme vidět na obrázku 5.1.



Obrázek 5.1: Snímek ze serveru Azure DevOps s výsledky testů

Zdroj: Azure DevOps Server [28]

Po dokončení testu můžeme zobrazit i protokol běhu testovací služby. Protokol je vytvořen v kořenové složce zkompilovaného testovacího projektu, tedy ve složce, ze které je testování spouštěno. Nejjednodušším způsobem, jak tento soubor zobrazit, je vypsáním jeho obsahu skrz příkazovou řádku. Zároveň lze vytvořit pravidlo, aby byl tento protokol vypisován pouze v případě neúspěchu nějakého z testů. Výpis protokolu z demonstrativního testovacího běhu můžeme vidět na výpisu 5.5.

```
class TestModdReadData : public ITestCase {
public:
    bool StartUp() {
        NV_IP_SUITE* pIpSuite;
        MFD_INT len;
        MFD_IP_SUITE networkInfo;
        MFD_AlignPack alignPack = MFD_ALIGN_PACK_1;

        // Configuration of ModbusTCP
        if ((mfd_modd_add_configuration(1, 0, 4, 2, 0, alignPack,
        ↪  MFD_MODULE_STANDARD)) != MFD_TRUE) {
            return false;
        }

        // Restore network information from memory
        Bsp_nv_data_restore(PNIO_NVDATA_IPSUITE, (PNIO_VOID**)&pIpSuite,
        ↪  (PNIO_UINT32*)&len);
        networkInfo.ipAddress = LSA_HTONL(pIpSuite->IpAddr);
        networkInfo.ipMask = LSA_HTONL(pIpSuite->SubnetMask);
        networkInfo.ipGateway = LSA_HTONL(pIpSuite->DefRouter);

        // ModbusTCP inicialization
        if (mfd_modd_online(&networkInfo) != MFD_TRUE) {
            return false;
        }

        if (mfd_modd_start() != MFD_TRUE) {
            return false;
        }

        // Set output data
        MFD_BYTE* data;
        MFD_BYTE control_data[8] = { 0,0xDE,0,0xAD,0,0xBE,0,0xEF };

        mfd_modd_out_data_lock(const_cast<const MFD_BYTE**>(&data));
        MFD_MEMCPY(data, control_data, 8);
        return mfd_modd_out_data_unlock() == MFD_TRUE;
    }

    bool Test() {
        return true;
    }

    bool TearDown() {
        return mfd_modd_stop() == MFD_TRUE && mfd_modd_offline() == MFD_TRUE
        ↪  && mfd_modd_remove_configurations() == MFD_TRUE;
    }
};
```

Výpis 5.2: Implementace testu na testovaném zařízení

```
class TestModdReadData : ITestCase
{
    public bool StartUp()
    {
        return true;
    }

    public bool Test()
    {
        // Get IP address of test device
        IPAddress devkit;
        if (!API.GetParticipantIp(PhysicalAddress.Parse("080006020110"), out
        ↪  devkit))
            return false;

        // Create ModbusTCP connection
        client = new ModbusClient(devkit.ToString(), 502);

        try
        {
            client.Connect();
        }
        catch
        {
            return false;
        }

        if (!client.Connected)
            return false;

        // Read data from device
        var test = client.ReadHoldingRegisters(0, 4);

        // Compare to expected data
        int[] controlData = { 0xDE, 0xAD, 0xBE, 0xEF };
        return Enumerable.SequenceEqual(controlData, test);
        }


    public bool TearDown()
    {
        client.Disconnect();
        return true;
    }

    ModbusClient client;
}
```

Výpis 5.3: Implementace testu pro testovacího partnera

```
[TestClass]
public class Modbus
{
    [TestMethod]
    public void TestModdAddConf()
    {
        API.Run(TestCaseE.MODD_ADD_CFG);
    }

    [TestMethod]
    public void TestModdReadData()
    {
        for (int x = 0; x < 8; x++)
            API.AddTestPartner(new TestLib.Client.TestPartner(new
            ↪  TestModdReadData()));

        API.Run(TestCaseE.MODD_READ_DATA);
    }

}
```

Výpis 5.4: Implementace testů v testovacím projektu

```
[4.5.21 - 14:55:26] Server ready on 192.168.0.15:1337
[4.5.21 - 14:55:26] Expected physical participants: 1
[4.5.21 - 14:55:35] Device with MAC address 080006020110 connected
[4.5.21 - 14:55:35] All devices connected, starting tests
[4.5.21 - 14:55:35] Running test: MODD_ADD_CFG
[4.5.21 - 14:55:35] Test successful
[4.5.21 - 14:55:35] Test partner connected
[4.5.21 - 14:55:36] Test partner connected
[4.5.21 - 14:55:36] Test partner connected
[4.5.21 - 14:55:36] Test partner connected
[4.5.21 - 14:55:36] Test partner connected
[4.5.21 - 14:55:36] Test partner connected
[4.5.21 - 14:55:36] Test partner connected
[4.5.21 - 14:55:36] Test partner connected
[4.5.21 - 14:55:36] Running test: MODD_READ_DATA
[4.5.21 - 14:55:38] Test successful
[4.5.21 - 14:55:38] Test partner disconnected
[4.5.21 - 14:55:38] Test partner disconnected
[4.5.21 - 14:55:38] Test partner disconnected
[4.5.21 - 14:55:38] Test partner disconnected
[4.5.21 - 14:55:38] Test partner disconnected
[4.5.21 - 14:55:38] Test partner disconnected
[4.5.21 - 14:55:38] Test partner disconnected
[4.5.21 - 14:55:38] Test partner disconnected
[4.5.21 - 14:55:38] Run finnished, all tests finnished successfully
```

Výpis 5.5: Protokol z provedeného testovací běhu

Kapitola 6

Vyhodnocení vytvořeného řešení

V této kapitole se věnuji zhodnocení vytvořeného řešení a jeho přínosu.

6.1 Bližší kontext

Vytvořená testovací knihovna míří na automatizaci testů verifikace průmyslové komunikace. Tato automatizace je následně využita při procesech kontinuální integrace a kontinuálního testování.

Tyto procesy přináší do vývoje softwaru několik výhod. Ze studie, která zkoumala dva porovnatelné projekty, kdy jeden z projektů používal kontinuální integraci a kontinuální testování oproti druhému projektu, víme, že za pomoci těchto procesů byl projekt schopen odhalit mnohem více chyb před vydáním softwaru oproti druhému projektu. Zároveň projekt bez zmíněných procesů potřeboval na odstranění chyb více času oproti druhému projektu. [29]

Toto je ještě více podstatné v kontextu testovaného produktu. Na industriální zařízení jsou kladeny mnohem větší požadavky na stabilitu a spolehlivost. Odhalení chyby při ostrém provozu může vést k ekonomické škodě zákazníka, ať už je to v lepším případě ztrátou zisku, a nebo v horším případě materiální škodou.

Důležitá je také implementace těchto procesů. Logickým cílem každé firmy je implementovat tyto procesy za co nejméně peněz a s minimální námahou. Přesně na tyto aspekty cílí vytvořená testovací knihovna.

6.2 Přínos vytvořeného řešení

Automatizované testy verifikace průmyslové komunikace vyžadují určitý počet komunikujících zařízení, která následně komunikují s testovaným produktem. Počet a typ zařízení se následně může měnit v závislosti na jednotlivých testovacích případech.

To ale vytváří vysoké hardwarové nároky na testování. Tedy k tomu, aby bylo možné automatizovaně testovat, je potřeba vytvářet sestavy zařízení, na kterých je následně testováno. Počet potřebných zařízení se dá teoreticky snížit v případě, že budou jednotlivé sestavy pravidelně přestavovány. Přestavováním se vytváří na zařízeních opotřebení a zároveň mohou vzniknout chyby. Přestavováním sestav by také testování nebylo už plně automatizované, proto je pro firmu výhodnější vytvořit více sestav v různých konfiguracích.

Nákup zařízení, která se podílejí na testovaní vyvíjeného projektu, jsou nemalou položkou v rozpočtu projektu. S využitím testovací knihovny ale můžeme tyto náklady v budoucnu snížit. Možnou úsporu zařízení si můžeme ukázat na jedné z investic do projektu. Seznam nakoupených zařízení můžeme vidět v tabulce 6.1. Seznam zařízení dobře reprezentuje významné výrobce používající průmyslové protokoly ModbusTCP a EtherNet/IP.

Současně je také potřeba zakoupit licence na software, který je potřebný k ovládání těchto zařízení. Seznam licencí potřebných k ovládaní zařízení v tabulce 6.1 můžeme vidět v tabulce 6.2. Všechny ceny zařízení a licencí jsou vzaty z volně dostupných nabídek. Jednotlivé zdroje lze nalézt v příloze B.

Využitím testovací knihovny a prostřednictvím testovacího partnera je ale možné tato zařízení simulovat. Simulací lze náklady na testování výrazně snížit. Cílem knihovny ale není použití fyzických zařízení úplně odstranit. Stále budou existovat případy, kdy tato zařízení budou stále potřeba.

Jak ale můžeme v tabulkách 6.1 a 6.2 vidět, spousta věcí je potřeba zajistit ve větším množství. Pokud za pomoci testovací knihovny snížíme počet zařízení tak, že odstraníme potřebu duplicitních zařízení, můžeme v rozpočtu projektu ušetřit až 252 767 Kč. Tím také snižujeme nároky na licence, jelikož méně zařízení vyžaduje méně lidí, kteří je současně konfigurují. Pokud budeme předpokládat, že počet potřebných licencí se taktéž stejně sníží, poté se v rozpočtu projektu může ušetřit až 130 034,64 Kč. Je rovněž nutné vzít v potaz další součásti, které jsou potřeba ke zprovoznění těchto zařízení. Využitím menšího množství zařízení můžeme zároveň ušetřit na ostatních materiálech, jako jsou ethernetové kabely atd. Jednotlivá zařízení můžou zároveň selhat, což poté vyžaduje jejich opětovnou koupi.

6.3 Výhody vytvořeného řešení

Při vytváření jednotlivých sestav je potřeba vzít i v potaz čas, který je potřeba ke stavbám, modifikacím a údržbě sestav a zařízení v nich. Toto se stává ještě obtížnější v době pandemie, kdy většina práce je prováděna z domova. Díky modulárnosti testovací knihovny lze jednoduše měnit jednotlivé konfigurace zařízení a tím snížit čas potřebný ke správám jednotlivých systémů.

Využitím testovací knihovny vznikají místo hardwarových problémů problémy softwarové, které se dají lépe podchytit a efektivněji řešit. Zároveň

použitím knihovny odstraňujeme limity jednotlivých zařízení, které jsou jinak udávány jejich výrobcem.

Velkou výhodou je též univerzálnost knihovny. Všechna zařízení v tabulce 6.1 využívají průmyslové protokoly ModbusTCP nebo EtherNet/IP, ale nic nebrání využití testovací knihovny s jinými průmyslovými protokoly. Díky tomu může být testovací knihovna využita i v ostatních projektech.

6.4 Nevýhody vytvořeného řešení

Nevýhodou vytvořeného řešení je jeho vysoká závislost na jednotlivých implementacích zařízení, respektive jednotlivých implementacích testovacích partnerů. Tyto testovací partnery je zapotřebí vyvinout, což znamená další časovou zátěž. Od těchto implementací se rovněž bude odvíjet celková kvalita testovací knihovny a limity jejího možného použití.

Čas vývoje je ale možné zkrátit, například využitím open-source knihoven pro jednotlivé průmyslové protokoly. Kvalita zařízení se dá též udržet jejich testováním a používáním nejnovějších verzí použitých open-source knihoven. Zároveň by produkt a testovací partner neměl být vyvíjen jedním člověkem.

6.5 Shrnutí

Testovací knihovna má vysoký potenciál výrazně snížit výdaje projektu na testovaní produktu. Výše úspor se však bude odvíjet od výše implementace jednotlivých testovacích partnerů a využití testovací knihovny.

Uvedená ukázka možných úspor poukazuje pouze na jednu investici do projektu a má demonstrativní charakter. Výše reálných úspor se může například zvýšit využitím testovací knihovny na více projektech.

Výrobce	Model	Množství	Cena za kus bez DPH	Celková cena bez DPH
Beckhoff	CX2030-0125	2	66 650,44 Kč	133 300,88 Kč
Beckhoff	CX2100-0004	2	8 663,21 Kč	17 326,42 Kč
Schneider Electric	BMXP342020 340-20	1	30 536,8 Kč	30 536,80 Kč
Schneider Electric	BMXCPS2000	1	5 220,27 Kč	5 220,27 Kč
Schneider Electric	BMXXBP0400	1	3 493,03 Kč	3 493,03 Kč
Schneider Electric	M251MESC	2	8 192,46 Kč	16 384,92 Kč
Schneider Electric	TM262L20MESE8T	1	36 636,86 Kč	36 636,86 Kč
Rockwell Automation	1769-L24ER-QB1B	2	38 526,54 Kč	77 053,08 Kč
Rockwell Automation	1783-BMS20CGN	2	64 480,27 Kč	128 960,54 Kč

Tabulka 6.1: Seznam nakoupených zařízení do projektu

Produkt	Množství	Cena za kus bez DPH	Celková cena bez DPH
Schneider Electric Machine Expert	2	23 987,18 Kč	47 974,36 Kč
Schneider Electric SO Machine	2	26 907,28 Kč	53 814,56 Kč
Rockwell Automation Studio 5000	2	79 140,18 Kč	158 280,36 Kč
Schneider Electric Unity Pro	1	102 299,92 Kč	102 299,92 Kč

Tabulka 6.2: Seznam licencí potřebných k ovládání nakoupených zařízení

Závěr

Cílem této práce bylo vytvořit testovací knihovnu, která umožní automatizovat testy verifikace průmyslové komunikace. Tato knihovna následně měla být propojena s Azure DevOps serverem tak, aby Azure DevOps server mohl automaticky testy spouštět a poskytovat uživateli informace o průběhu jednotlivých testů. Dalším cílem práce bylo prozkoumat dostupné open-source knihovny pro průmyslové protokoly ModbusTCP a EtherNet/IP a za pomoci jedné z knihoven demonstrovat funkcionalitu vytvořeného řešení. V neposlední řadě bylo cílem zhodnotit vytvořené řešení a jeho přínosy.

Cíle této práce byly splněny. Práce v kapitole 2 definuje jednotlivé pojmy a přibližuje kontext této práce. Testovací knihovna a všechny její části byly následně v kapitole 3 navrhnuty. Testovací služba, která řídí testovací běh, byla navrhnuta jako doplněk do testovacího frameworku MSTest. Následně za pomoci tohoto frameworku bylo řešení propojeno se serverem Azure DevOps, který mohl jednotlivé testy registrovat, automaticky spouštět a zobrazovat jejich výsledky. Implementace všech částí testovací knihovny byla následně popsána v kapitole 4 a funkčnost vytvořeného řešení demonstrována v kapitole 5. Kapitola 6 následně zhodnotila vytvořené řešení, jeho přínosy a úskalí z pohledu projektového managementu.

Vytvořené řešení splňuje požadavky, které na něj byly kladeny, avšak je zde stále prostor pro zlepšení. Zřejmým rozšířením knihovny je implementace jednotlivých testovacích partnerů, kteří budou simulovat chování jednotlivých zařízení. Zároveň je možné zlepšit registraci testů, respektive automatizovat tuto registraci, na testovaném zařízení.

Bibliografie

1. SINGH, Sanjay Kumar; SINGH, Amarjeet. *Software testing*. Vandana Publications, 2012.
2. FEWSTER, M.; GRAHAM, D. *Software test automation*. Addison-Wesley Reading, 1999.
3. KITCHENHAM, B.; PFLEEGER, S. L. Software quality: the elusive target [special issues section]. *IEEE Software*. 1996, roč. 13, č. 1, s. 12–21. Dostupné z DOI: 10.1109/52.476281.
4. LUO, Lu. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*. 2001, roč. 15232, č. 1-19, s. 19.
5. KHAN, Mohd Ehmer. Different forms of software testing techniques for finding errors. *International Journal of Computer Science Issues (IJCSI)*. 2010, roč. 7, č. 3, s. 24.
6. SHAHIN, M.; ALI BABAR, M.; ZHU, L. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*. 2017, roč. 5, s. 3909–3943. Dostupné z DOI: 10.1109/ACCESS.2017.2685629.
7. *Plan, code, collaborate, ship applications - Azure DevOps* [online]. 2021 [cit. 2021-04-07]. Dostupné z: https://docs.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops?view=azure-devops.
8. *Collaborate on code - Azure Repos* [online]. 2020 [cit. 2021-04-07]. Dostupné z: https://docs.microsoft.com/en-us/azure/devops/repos/get-started/what-is-repos?view=azure-devops.

9. *What is Azure Pipelines? - Azure Pipelines* [online]. 2019 [cit. 2021-04-07]. Dostupné z: https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops.

10. *Exploratory and manual testing overview - Azure Test Plans* [online]. 2019 [cit. 2021-04-07]. Dostupné z: https://docs.microsoft.com/en-us/azure/devops/test/overview?view=azure-devops.

11. *Artifacts in Azure Pipelines - Azure Pipelines* [online]. 2021 [cit. 2021-04-07]. Dostupné z: https://docs.microsoft.com/en-us/azure/devops/pipelines/artifacts/artifacts-overview?view=azure-devops&tabs=nuget.

12. *Azure Pipelines Agents* [online]. 2020 [cit. 2021-03-15]. Dostupné z: https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/agents?view=azure-devops.

13. SHETH, Himanshu. *Most Complete MSTest Framework Tutorial Using .Net Core* [online]. 2021 [cit. 2021-04-07]. Dostupné z: https://www.lambdatest.com/blog/most-complete-mstest-framework-tutorial-using-net-core-2/.

14. THOMESSE, J. P. Fieldbus Technology in Industrial Automation. *Proceedings of the IEEE*. 2005, roč. 93, č. 6, s. 1073–1101. Dostupné z DOI: 10.1109/JPROC.2005.849724.

15. WOLLSCHLAEGER, M.; SAUTER, T.; JASPERNEITE, J. The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things and Industry 4.0. *IEEE Industrial Electronics Magazine*. 2017, roč. 11, č. 1, s. 17–27. Dostupné z DOI: 10.1109/MIE.2017.2649104.

16. LEE, K. C.; LEE, S. Performance evaluation of switched Ethernet for real-time industrial communications. *Computer Standards & Interfaces*. 2002, roč. 24, č. 5, s. 411–423. ISSN 0920-5489. Dostupné z DOI: https://doi.org/10.1016/S0920-5489(02)00070-3.

17. MODBUS ORGANIZATION. *Modbus Application Protocol Specification v1.1b3* [online]. 2012 [cit. 2021-04-07]. Dostupné z: https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf.

18. ZEZULKA, F.; HYNČICA, O. Průmyslový Ethernet IX: Ethernet/IP, EtherCAT. *Automa*. 2008, roč. 9, č. 10, s. 60–64. Dostupné také z: https://automa.cz/Aton/FileRepository/pdf_articles/37910.pdf.

19. *SIMATIC ET 200* [online]. 2021 [cit. 2021-04-09]. Dostupné z: https://assets.new.siemens.com/siemens/assets/api/uuid:efaefc8d-c213-4ea6-bd48-43efd20ca566/dffa-b10058-01-7600.pdf.

20. *SIMATIC ET 200SP - The powerful IO system for compact control cabinets* [online]. 2021 [cit. 2021-04-09]. Dostupné z: https://new.siemens.com/global/en/products/automation/systems/industrial/io-systems/et-200sp.html.

21. *Maximum packet size for a TCP connection* [online]. 2010 [cit. 2021-04-07]. Dostupné z: https://stackoverflow.com/questions/2613734/maximum-packet-size-for-a-tcp-connection.

22. *Attributes (C#)* [online]. 2018 [cit. 2021-03-22]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/.

23. MICROSOFT CORPORATION. *Visual Studio Professional 2019 verze 16.4.19* [software]. 2021 [cit. 2021-05-06].

24. *Test Result XML Format* [online]. 2018 [cit. 2021-04-24]. Dostupné z: https://docs.nunit.org/articles/nunit/technical-notes/usage/Test-Result-XML-Format.html.

25. BARRÉ, Gérald. *MSTest v2: Test lifecycle attributes - Gérald Barré* [online]. 2018 [cit. 2021-03-28]. Dostupné z: https://www.meziantou.net/mstest-v2-test-lifecycle-attributes.htm.

26. *Unit testing C# with MSTest and .NET Core - .NET* [online]. 2020 [cit. 2021-03-28]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-mstest.

27. *Assemblies in .NET* [online]. 2019 [cit. 2021-03-28]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/assembly/.

28. MICROSOFT CORPORATION. *Azure DevOps Server verze Dev18.M170.8* [software]. 2020 [cit. 2021-05-06].

29. AMRIT, Chintan; MEIJBERG, Yoni. Effectiveness of Test-Driven Development and Continuous Integration: A Case Study. *IT Professional*. 2018, roč. 20, č. 1, s. 27–35. Dostupné z DOI: 10.1109/MITP.2018.014121554.

Příloha **A**

Seznam použitých zkratek

ADU Application Data Unit

API Application Programming Interface

CIP Common Industrial Protocol

EtherNet/IP EtherNet Industrial Protocol

I/O Input/Output

IDE Integrated Development Environment

IP Internet Protocol

PDU Protocol Data Unit

PLC Programmable Logic Controller

TCP/IP Transmission Control Protocol/Internet Protocol

XML Extensible markup language

Příloha B

Seznam zdrojů cen zařízení a licencí

V tabulce B.1 lze vidět seznam zařízení, jejich cenu a zdroj ceny. Ceny byly získány 16.04.2021 a případně převedeny do českých korun dle kurzu k tomuto dni.

Produkt	Cena za kus bez DPH	Zdroj
Beckhoff CX2030-0125	66 650,44 Kč	https://www.radwell.co.uk/Buy/BECKHOFF/BECKHOFF/CX2030-0125
Beckhoff CX2100-0004	8 663,21 Kč	https://www.radwell.co.uk/en-GB/Buy/BECKHOFF/BECKHOFF/CX2100-0004/
Schneider Electric BMXP342020 340-20	30 536,8 Kč	https://cz.farnell.com/schneider-electric/bmxp342020/processor-module-0-095a-24vdc/dp/2835381
Schneider Electric BMXCPS2000	5 220,27 Kč	https://cz.rs-online.com/web/p/prislusenstvi-pro-plc/0148026/
Schneider Electric BMXXBP0400	3 493,03 Kč	https://uk.rs-online.com/web/p/plc-accessories/0147922/
Schneider Electric M251MESC	8 192,46 Kč	https://cz.rs-online.com/web/p/plc-programovatelne-logicke-kontrolery/8066755/
Schneider Electric TM262L20MESE8T	36 636,86 Kč	https://at.rs-online.com/web/p/sps-zentralbaugruppen/2011448/
Rockwell Automation 1769-L24ER-QB1B	38 526,54 Kč	https://www.routeco.com/en-gb/shop/programmable-controllers/compactlogix/1769-l24er-qb1b
Rockwell Automation 1783-BMS20CGN	64 480,27 Kč	https://cz.wiautomation.com/allen-bradley/industrial-communication/stratix/1783BMS20CGN
Schneider Electric Machine Expert	23 987,18 Kč	https://cz.wiautomation.com/schneider-electric/software/ESECAPCZZSPMZZ
Schneider Electric SO Machine	26 907,28 Kč	https://www.alliedelec.com/product/schneider-electric/somnacczxspazz/70596759
Rockwell Automation Studio 5000	79 140,18 Kč	https://twcontrols.com/lessons/where-can-i-download-studio-5000-logix-designer-rslogix-5000-and-what-is-the-pricedifference-of-each-version
Schneider Electric Unity Pro	102 299,92 Kč	https://cz.wiautomation.com/schneider-electric/software/UNYUPDSPUXUG

Tabulka B.1: Seznam zdrojů cen zařízení a softwarových licencí

Příloha C

Obsah přiložené SD karty

- `README.txt` ..... stručný popis obsahu SD karty
- `impl` ..... zdrojové kódy implementace
  - `cpp` ..... implementace pro testované zařízení v jazyku C++
  - `core` ..... implementace jádra testovací knihovny v jazyku C#
  - `test_project` ... testovací projekt vytvořený k demonstraci funkčnosti
- `package` ..... adresář se zkompilovaným balíčkem testovací knihovny
- `text` ..... text práce
  - `src` ..... zdrojová forma práce ve formátu LaTeX
  - `thesis.pdf` ..... text práce ve formátu PDF