VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2020 Bc. Tomáš Kohoutek

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

EXPERIMENTÁLNÍ APLIKACE ROBOTICKÉ PAŽE VYUŽÍVAJÍCÍ PRŮMYSLOVÉ KOMUNIKAČNÍ PROTOKOLY
EXPERIMENTAL APPLICATION OF ROBOTIC ARM USING INDUSTRIAL COMMUNICATION PROTOCOLS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Tomáš Kohoutek**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Radek Fujdiak, Ph.D.**
SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**

Ústav telekomunikací

Student: Bc. Tomáš Kohoutek ***ID:*** 125245
Ročník: 2 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Experimentální aplikace robotické paže využívající průmyslové komunikační protokoly

POKYNY PRO VYPRACOVÁNÍ:

Student bude mít za úkol seznámit se s prostředím Dobot Magician a s příslušnou robotickou paží. Následně navrhnout a implementovat průmyslovou smyčku (zacyklený úkol) odpovídající reálnému provozu. Následně bude k robotovi připojena řídící jednotka (např. RaspBerry Pi 3B+), která bude mít za úkol posílat řídící příkazy přímo na robota. Řídící jednotka bude obsahovat „firmware“ resp. mezivrstvu (middleware) mezi průmyslovými protokoly a samotným robotem. Mezivrstva bude otestována na vybraných průmyslových protokolech (alespoň dva) a následně bude zprovozněna reálná průmyslová síť složená z výše uvedených komponentů. Výsledkem tak bude finální řešení simulující reálný provoz v zacyklené smyčce využívající průmyslové protokoly.

DOPORUČENÁ LITERATURA:

[1] RÜßMANN, Michael, et al. Industry 4.0: The future of productivity and growth in manufacturing industries. Boston Consulting Group, 2015, 9.1: 54-89.

[2] GILCHRIST, Alasdair. Industry 4.0: the industrial internet of things. Apress, 2016.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 1.6.2020

Vedoucí práce: Ing. Radek Fujdiak, Ph.D.

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Diplomová práce je zaměřena na analýzu nejznámějších průmyslových protokolů a seznámením se s robotickou paží Dobot Magician. Hlavním cílem této práce je tvorba průmyslové smyčky, která je tvořena komunikací – klient a server. Jednotlivé komponenty průmyslové smyčky jsou tvořeny třemi robotickými pažemi (včetně periferií), jejich řídícími jednotkami (Raspberry Pi 3B+), přepínačem a serverem. Řídící jednotky komunikují se serverem za pomocí dvou vybraných průmyslových protokolů a to Modbus TCP a EtherNET/IP. Výsledkem práce je vytvořená nekonečná smyčka, která se snaží podobat reálnému provozu skladby a rozkladu krabičky obsahující různé předměty.

KLÍČOVÁ SLOVA

Chytrá továrna, Dobot Magician, Modbus TCP, EtherNET/IP, pydobot, Raspberry Pi, pymodbus, cpppo

ABSTRACT

This diploma thesis is focused on the analysis of the most well-known industrial protocols and acquaintance with the Dobot Magician robotic arm. The main goal of this thesis is to create an industrial loop, which consists of communication – client and server. The individual components of the industrial loop consist of three robotic arms (including peripherals), their control units (in our case Raspberry PI 3B +), switch, and a server. The control units communicate with the server using two selected industrial protocols, namely Modbus TCP and EtherNET/IP. The result of the work is an endless loop, which tries to resemble the real operation of the assembly and disassembly of the box equipped with different items.

KEYWORDS

Smart Factory, Dobot Magician, Modbus TCP, EtherNET/IP, pydobot, Raspberry Pi, pymodbus, cpppo

KOHOUTEK, Tomáš. *Experimentální aplikace robotické paže využívající průmyslové komunikační protokoly*. Brno, Rok, 110 s. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. Radek Fujdiak, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Experimentální aplikace robotické paže využívající průmyslové komunikační protokoly“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Radku Fujdiakovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. Dále bych rád poděkoval panu Ing. Ondřeji Pospíšilovi za trpělivost a dohled nad mou osobou při realizaci úlohy v laboratoři. V poslední řadě bych rád poděkoval celé mé rodině a přítelkyni za podporu v těchto nelehkých časech.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Prolínání informačních a průmyslových technologií je v dnešní době velice diskutované téma. Hlavním tématem této diplomové práce je aplikace a testování těchto dvou navzájem prolínajících se technologií. Jako jedna z možností se jeví vhodné použití robotické paže firmy Dobot.cc, která je dostupná jak funkčně, tak i cenově pro širokou veřejnost. Tímto se tak naskytuje příležitost využití průmyslového zařízení v běžných podmínkách i mimo tovární prostředí a široká škála možnosti, jak simulovat její funkce.

V úvodních kapitolách diplomové práce je představeno prostředí pro chytrou továrnu, a jeho dopad na konvergenci mezi IT a OT. Dále budou čtenáři seznámeni se systémy a nejčastějšími periferiemi používanými v tomto prostředí. V dalších kapitolách budou rozebrány známé průmyslové protokoly, jako je Modbus, PROFINET, EtherNET/IP a DNP3 včetně jejich historie a způsobu komunikace.

V praktické části se podíváme blíže na dostupnou robotickou paži s názvem Dobot Magician včetně volby dostupných knihoven pro implementaci a tvorbu testovacího prostředí. Dále je práce zaměřena na instalaci potřebných dílčích součástí jak na straně serveru tak, klienta. Následně bude otestována funkční komunikace a vytvořen a řádně zdokumentován program pro robotickou paží Dobot Magician. Rovněž bude patřičně zdokumentována analýza programem Wireshark.

Po otestování základní funkčnosti v testovacím prostředí bude vytvořena průmyslová smyčka ze všech dostupných komponent. Průmyslová smyčka bude obsahovat tři robotické paže Dobot Magician, kdy každá bude obsluhovat jinou periferii. Robotická paže označena jako ID1 obsluhuje kolejnici, robotická paže označena jako ID2 bude obsluhovat senzor barev a robotická paže označena jako ID3 bude ovládat pásový dopravník. Následně bude průmyslová smyčky popsána jak ze strany serveru, tak ze strany klientů, kdy klient s ID1 a ID2 komunikují se serverem za pomocí protokolu Modbus TCP a klient s ID3 za pomocí protokolu EtherNET/IP. Nakonec bude opět provedena analýza komunikace rozborem záznamu z programu Wireshark.

1 Princip chytré továrny

Vzhledem k tomu, že se technologie po celém světě neustále vyvíjí, tak výrobci čím dál tím více přijímají koncepci chytré továrny. Jedná se o požadavek mnoha odborníků z oblasti techniky a průmyslu, kteří se domnívají, že optimalizace výrobního procesu za účelem tvorby integrovaného a kolaborativního procesu přinese další průmyslovou revoluci na světě. Národní institut pro standardy a technologie uvedl, že chytrá továrna je: „plně integrované výrobní systémy založeny na spolupráci, která reaguje v reálném čase na splnění měnících se požadavků a podmínek v továrně, v dodavatelské síti a v potřebách zákazníků“ [1]. Chytrá továrna představuje posun vpřed od tradičních automatizací k plně připojenému a flexibilnímu systému, který může využívat neustálý tok dat z připojených provozních a výrobních systémů k učení a přizpůsobení se novým požadavkům.

1.1 Definice chytré továrny

Automatizace byla vždy do určité míry součástí továrny a ani vysoká úroveň automatizace není žádným novým pojmem. Pojmem „automatizace“ však není myšlen výkon jednoho diskrétního úkolu nebo procesu. Z historického hlediska byly situace, kdy stroje učinily „rozhodnutí“, založené na automatizaci a linearitě, jako je například otevření ventilu nebo zapnutí či vypnutí čerpadla na základě předem definované sady instrukcí a pravidel. Za pomocí využití umělé inteligence (AI) a zvyšující se sofistikovanosti kyberfyzikálních systémů, které mohou kombinovat jak fyzické stroje a obchodní procesy, automatizace stále více zahrnuje složitá optimalizační rozhodnutí, která lidé obvykle činí [2].

Nakonec a možná nejdůležitější pojem „chytrá továrna“ také navrhuje integraci rozhodování a poznatků z prostředí výroby do zbytku dodavatelského řetězce a širšího podniku za pomocí propojeného prostředí IT/OT. To může v konečném hledisku zásadně změnit výrobní procesy a zlepšit tak vztahy s dodavateli a zákazníky.

Z předešlého popisu je nyní zřejmé, že chytré továrny vysoce přesahují jednoduchou automatizaci. V případě chytré továrny se jedná o flexibilní systém, který dokáže optimalizovat výkon v širší síti, přizpůsobovat se novým podmínkám a učit se od nich v reálném čase nebo v reálném čase se zpožděním a samostatně provozovat celé výrobní procesy [3]. Chytré továrny mohou pracovat pouze uvnitř jedné výrobní haly a mohou také být navzájem propojeny za pomocí celosvětové sítě podobných výrobních systémů a tím tvořit komplexní celek.

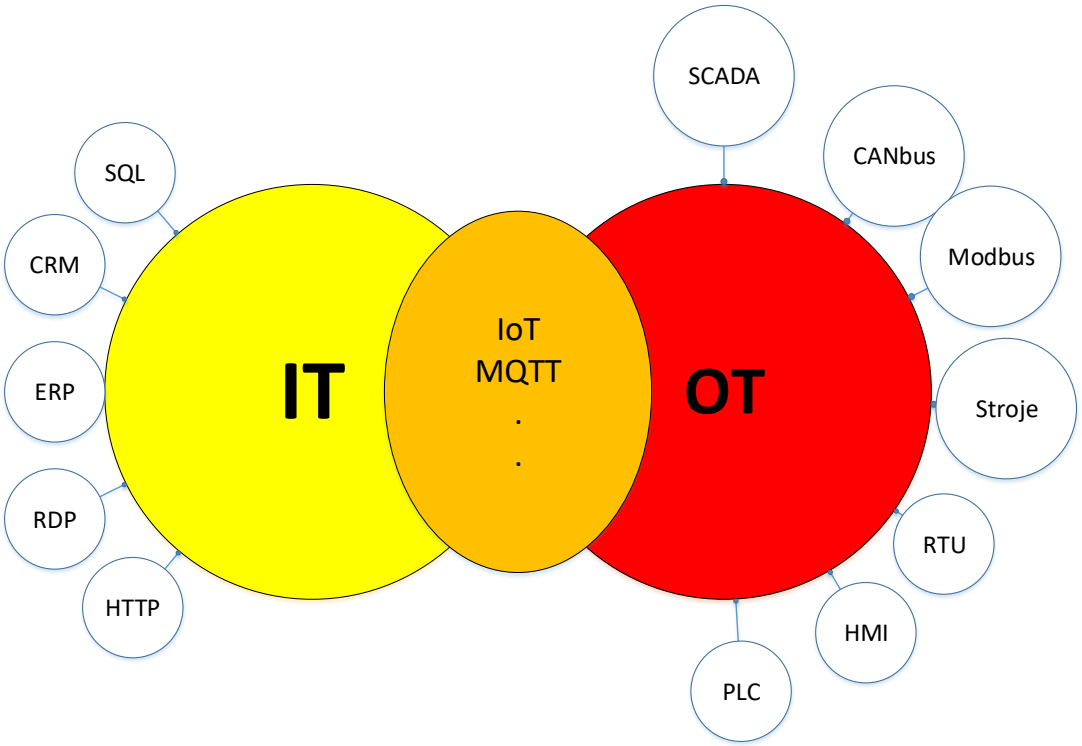
Je však velice důležité si uvědomit, že chytrá továrna tak, jak je zde definována a popsána, by neměla být považována za „finální stav“ vzhledem k velice rychlému

tempu technologického rozvoje. Koncept představuje spíše neustálý vývoj a nepřetržitou cestu k budování a udržování pružného systému, nežli, jak v minulosti docházelo pouze k modernizaci továrny třeba výměnou jednoho stroje. Skutečná přednost chytré továrny spočívá v její schopnosti se vyvíjet a růst spolu s měnícími se potřebami organizace, ať už v závislosti na změně poptávky od zákazníka, vývoj nových produktů, služeb, začlenění nových procesů a technologií nebo expanzi na nové trhy [4]. Za pomocí výkonnějším a analytickým schopnostem, spolu s širšími ekosystémy inteligentních prostředků se mohou chytré továrny změnám přizpůsobovat mnohem snadněji než v minulosti kdy by proces přizpůsobení byl velice složitý ne-li nemožný.

2 Konvergence OT s IT

Dříve byly informační technologie (IT) odděleny od provozních technologií (OT), ale v poslední době můžeme vidět, jak se tyto dva světy navzájem prolínají. Nejenže se stávají stále více propojeny než dříve, ale rozšiřuje se i jejich konektivita k internetu. Tímto krokem mezi sebou mohou sdílet spoustu potřeb, problémů a zkušeností čímž vzniká větší efektivita, synergie a učení.

Aby bylo možné pokračovat je potřeba si vysvětlit rozdíly mezi IT a OT. Jak již je patrné ze zkratky IT, jakožto reprezentující informační technologie které jsou navázány na veškeré výpočetní systémy, jako jsou třeba SQL, CRM, ERP, RDP atd. Napříč tomu OT zastřešující provozní technologie jakožto hardware a software který detekuje nebo řídí změny skrz monitorování nebo ovládání fyzických strojů a zařízení. Příklady OT můžeme uvést RTU, PLC, SCADA, HMI a mnoho dalších které můžeme pozorovat na obrázku 2.1.



Obr. 2.1: Konvergence IT s OT.

2.1 Rozdíly mezi OT a IT

Pokud se v dnešní době použije výraz IT, každý má zběžnou představu, o co se zhruba jedná, pokud se ovšem použije zkratka OT, už málokdo má představu, co si pod provozními technologiemi představit. OT monitorují a řídí aktiva průmyslových procesů a dále potom výrobní/průmyslové vybavení. OT existuje mnohem déle než IT, konkrétně se může datovat do období, kdy jsme začali využívat stroje a zařízení poháněné elektrickým proudem v továrnách, budovách a dopravních systémech a mnoho dalších. Za OT můžeme považovat hardware a software, který udržuje továrny, elektrárny a vybavení v nich v chodu [5, 6].

2.1.1 Koncové stanice

- **OT:** Zjednodušeně řečeno „věci“, nechť je to třeba pumpa, ventil, senzor. Předmět, který není běžně spravován přímo lidmi. Stáří může být počítáno na dekády nebo delší časové intervaly. Ovládání je velice často automatizované a většinou lidská intervence souvisí pouze s údržbou nebo výměnou.
- **IT:** Nejčastěji výpočetní zařízení používané lidmi, jako jsou třeba notebooky, mobily a tablety. Většinou je správa pod přímou kontrolou příslušné osoby. Většina zařízení je nová ve srovnání ke staří OT, životnost je periodická od 3–5 let. Ovládání zařízení nebývá většinou automatické a je pod přímou správou uživatele [6].

2.1.2 Architektura

- **OT:** Spleť většinou uzavřených systémů se zaměřením na bezpečnost, pružnost, nadbytek a zmírnění rizika a tak dále. Systém bývá řízen proprietárně, zahrnuje proprietární standardy komunikace. Adaptace na nové technologie je většinou velice náročná.
- **IT:** Mnohem více standardizované a homogenní než OT architektura připojena za pomocí serverů, routerů a výchozích bran. Proprietární systémy existují, ale rychleji se adaptují na nové standardizace a inovace. Mnohem rychlejší adaptace a převzetí nových technologií [6].

2.1.3 Aplikace

- **OT:** Více věcně zaměřeného softwaru zprostředkovávajícího data a ovládání pro produkci nebo pohyb zboží a kontrolu systému. Lidská součinnost je většinou pouze z pozice supervizora s občasnou interakcí.

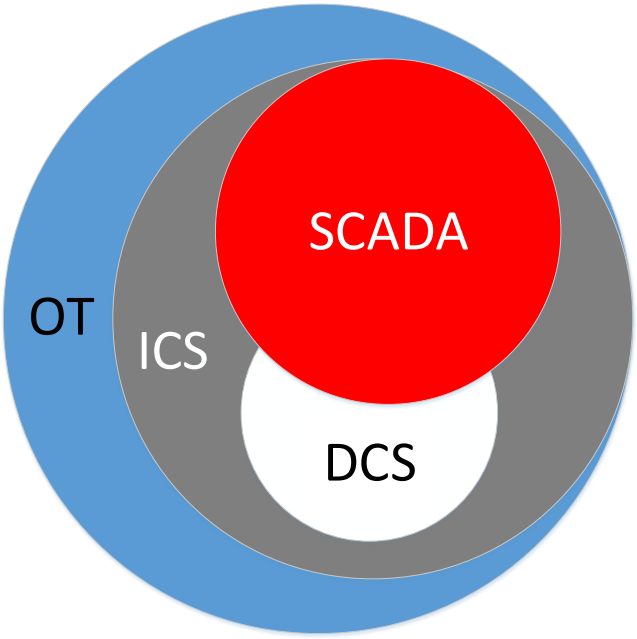
- **IT:** Více lidsky zaměřený software poskytující data pro jednotlivce pro tvorbu obchodních rozhodnutí. Lidé jsou doslova zapojeni do činění rozhodnutí.[6].

2.1.4 Rozsah a vlastnictví

- **OT:** Účelově vytvořené systémy dizajnované, vyvíjené a podporované pro přímé účely. Další vynaložené prostředky jsou využity pro vylepšení aplikací specifických pro daný úkol.
- **IT:** Podpora korporativního fungování jakožto lidské zdroje, finance, služby, CRM a mnoho dalších. Další finanční prostředky jsou vynaloženy pro korporátní obchodní procesy [6].

2.2 Řídící systém průmyslových systémů – ICS

ICS jakožto systém průmyslového řízení je termín, který se používá k popisu různých typů řídících systémů a přidruženého vybavení, která zahrnují zařízení, systémy, sítě a ovládací prvky využívané k provozu a nebo automatizaci průmyslových procesů. V závislosti na odvětví funguje každý ICS jinak, a je vytvořen tak, aby spravoval elektronicky úkoly efektivně. V dnešní době se zařízení a protokoly používané v ICS využívají téměř ve všech průmyslových odvětvích a v kritické infrastruktuře, jako je výroba, energetika, doprava a úprava vod [7].



Obr. 2.2: ICS jakožto součást OT.

Existuje mnoho typů ICS, z nichž nejčastěji využívané jsou systémy dohledové kontroly a získávání dat (SCADA) a distribuované řídící systémy (DCS). Lokální operace jsou často řízeny takzvanými field-devices, což mohou v praxi být pohony, polohovací prvky, frekvenční měniče atd., které přijímají dohledové příkazy od vzdálených stanic. Je potřeba si uvědomit, že ICS je stále součástí většího celku OT jak můžeme vidět na obrázku 2.2 a také, že implementace ICS bývá často hybridní kombinace systémů DCS a SCADA, kde mezi sebou systémy spolupracují.

2.2.1 Dispečerské řízení a sběr dat – SCADA

V první řadě je si potřeba říct, že SCADA není systém, který poskytuje plnou kontrolu. Místo toho je jeho funkčnost zaměřena na poskytování kontroly na úrovni dohledu. Systémy SCADA se skládají ze zařízení (obvykle se jedná o programovatelné logické kontroléry – PLC) v kombinaci s Remote Terminal Unit – RTU nebo jiných komerčních hardwarových modulů, které jsou distribuovány na různých místech. Systémy SCADA mohou získávat a přenášet data a jsou integrovány s rozhraním Human Machine Interface (HMI), které potom poskytuje centralizované monitorování a řízení spousty procesních vstupů a výstupů.

Hlavním účelem použití SCADA je sledování na dálku a řízení linek sítě prostřednictvím centralizovaného systému řízení. Namísto pracovníků, kteří by museli cestovat na větší vzdálenosti, aby mohli provádět stejné úkoly a nebo shromažďovat data, je SCADA systém schopen tento proces zcela automatizovat. Lokální zařízení řídí místní operace, jako je otevíraní nebo zavíraní ventilů popřípadě jističů, sběr dat ze senzorických systémů a monitoring místního prostředí z hlediska bezpečnostních podmínek [8].

2.2.2 Distribuovaný kontrolní systém – DCS

V případě DCS jde o systém, který se využívá k řízení výrobních systémů, které se nacházejí na jednom místě. V DCS je žádaná hodnota zaslána do řídící jednotky, která je schopna dát ventilům nebo dokonce akčnímu členu pokyn, aby pracoval takovým způsobem, aby byla zachována požadovaná hodnota. Data z oblasti využití mohou být uložena pro budoucí využití, použita pro jednoduché řízení procesů nebo použita pro pokročilé strategie řízení s daty z jiné oblasti zařízení.

Každý DCS používá centralizovanou smyčku pro správu více lokálních řadičů nebo zařízení, která jsou součástí celkového výrobního procesu. Díky tomu mají průmyslová odvětví rychlý přístup k provozním a výrobním údajům. Tím, že je využito více zařízení v rámci výrobního procesu je DCS schopen snížit dopad jediné chyby na celkový běh systému. DCS je běžně využíván v průmyslových odvětvích, jako je výroba, elektrárny, ropné rafinerie a další [9].

2.3 PLC – Programmable logic controller

Programovatelný logický automat nebo-li PLC je dnes všudypřítomný v procesním i výrobním průmyslu. PLC, které bylo původně vytvořeno jako náhrada za elektromagnetické reléové systémy, nabízí snadnější řešení pro úpravu provozu řídícího systému. Velice rychlé stahovaní z počítače nebo programovacího zařízení umožňuje logické změny řízení během několika minut nebo sekund.

PLC je průmyslový digitální počítač navržený k provádění řídících funkcí zejména pro průmyslové aplikace. Většina PLC je dnes modulární, což umožňuje uživateli přidat nebo odebrat sortiment funkcí včetně diskrétního řízení, analogového řízení, PID řízení, řízení motorů, řízení polohy, vysokorychlostního připojení k síti nebo sériové komunikace [10].

2.3.1 Základní komponenty PLC

PLC se skládá z několika základních částí, mezi ně patří napájení, centrální procesorová jednotka (CPU), vstupní/výstupní karty (I/O). PLC mohou být plně integrované v jednom zařízení kupovány jako celek anebo modulární (lze přidávat požadované moduly).

1. CPU modul je složen z centrálního procesoru a jeho paměti. Procesor je zodpovědný za provádění výpočtů a zpracovaní dat přijetím vstupů a vytvoření příslušných výstupů.
2. Modul napájení zprostředkovává požadovaný výkon do celého systému převodem dostupného střídavého proudu na stejnosměrný výkon potřebný pro moduly CPU a I/O.
3. I/O u PLC se používají k připojení senzorů a akčních členů k systému, kde snímají různé parametry, jako je teplota, tlak, atd. I/O moduly mohou být buď to analogové nebo digitální.

Programovatelný logický automaty jsou odlišné od běžných počítačů tím, že program je zpracováván cyklicky a jejich periferie jsou uzpůsobeny tak, aby mohli být napojeny přímo na technologické procesy [10].

2.3.2 Zpracování programu

Programovatelný logický automat pracuje ve čtyřech základních krocích, a to skenovaní vstupů, skenovaní programu, skenovaní výstupů, předávání instrukcí a komunikace.

1. **Skenovaní vstupů:** V prvním kroku je zjištěn stav všech vstupních zařízení, která jsou k PLC připojena.
2. **Skenovaní programu:** Ve druhém kroku provede program, který byl vytvořen uživatelem pro danou úlohu.
3. **Skenovaní výstupů:** Ve třetím kroku aktivuje napájení na výstupech dle instrukcí programu.
4. **Předávání instrukcí:** Čtvrtý krok zahrnuje komunikaci s terminály, diagnostikou, atd.

2.3.3 Programovací jazyky PLC

Na základě standardu Mezinárodní elektronické komise (IEC) dle normy IEC EN 61131-3 jsou programovací jazyky PLC klasifikovány na pět hlavních.

1. **Jazyk příčkového diagramu (LD)** – LD reprezentující Ladder diagram nebo je taktéž známý jako „Ladder Logic“. Je nejpoužívanějším programovacím jazykem pro PLC, tento jazyk se snadno učí. Pokud se podíváme na strukturu LD bude nám připomínat schéma elektrického obvodu, které napodobuje mechanické relé v panelu, které automat nahrazuje. LD je jednoduchá logická konstrukce a je spolehlivější než elektronický obvodový řadič. Program se snadno čte a učí, každý programovací symbol provádí specifické úkony. Výborně reprezentuje diskrétní logiku [11].
2. **Jazyk seznamu instrukcí (IL)** – IL používá mnemotechnický kód. Jedná se o textový jazyk, kdy kód programu se sestavuje z textových zkratek a několika základních instrukcí. Je obdobou programovacího jazyka assembler pro programování mikroprocesorů a mikrořadičů [11, 12].
3. **Jazyk strukturovaného textu (ST)** – Pro jazyk strukturovaného textu se využívají zkratky „ST“ nebo „STX“. Jazyk využívá syntaxi na úrovni vyššího programovacího jazyka – můžeme jej přiřadit k programování v Pascalu, využívají se smyčky, cykly, proměnné, podmínky a operátory. Mezi hlavní klady platí jeho srozumitelnost pro programátory a díky svému kódovému formátu lze snad upravovat [11, 12].
4. **Jazyk funkčního blokového schématu (FBD)** – Jazyk FBD neboli jazyk funkčního blokového schématu je grafický jazyk, který obsahuje obdélníkové značky funkcí a funkčních bloků, které jsou propojeny spojnicemi (jinak řečeno spojovacími vodiči). Jazyk FBD je vhodný pouze pro úkony do určité složitosti. Pracuje se s algoritmy logického, ale i numerického typu. Výhodou FBD je, že obvykle je sestavován z „prefabrikátů“ – funkčních bloků, ty mohou být použity z knihoven standardních funkčních bloků nebo přímo nakoupených od výrobce daného PLC [11, 12].

5. **Sekvenční funkční diagram (SFC)** – Jako u FBD je rovněž SFC grafickým jazykem, používaným pro sekvenční programování PLC. Forma programu je interpretována přechodovým grafem programu, který modeluje sekvenční chování [11, 12]. Jedná se o zobecnění Mooreova automatu v kombinaci s Petriho sítí.

2.4 Rozhraní člověk-stroj – HMI

V dnešní době je velice důležité vědět, co je to HMI. Rozhraní člověk-stroj je jedním z nejdůležitějších témat průmyslové automatizace. Zdá se, že právě budoucnost této oblasti leží v rukou této technologie. Mezi společnostmi došlo k tvrdému konkurenčnímu boji, aby byla vydána co nejinovativnější platforma, která bude schopna propojit tyto dva světy.

Ke komunikaci dochází převedením velkého množství komplexních dat na dostupné informace. Tímto způsobem má operátor všechny dostupné nástroje pro řízení výrobního procesu. Po kontextualizování této definice v oboru průmyslové automatizace je jasné, že čím víc vstřícnější a uživatelky přívětivější prostředí HMI, tím bude efektivnější práce [13].

2.4.1 Vývoj HMI

Ke skutečné definici HMI je nejprve udělat krok zpět. Je potřeba se podívat na to jak se tato technologie vyvinula. Vše začalo tlačítkem, následně se světlem přišly spínače, elektronické panely, dotykové displeje a nakonec s časem osobních počítačů a softwarových programů prošel svět rozhraní člověk-stroj obrovským vývojem.

Ale co činí tyto systémy potenciálně neomezenými? Jsou zde dvě odpovědi. Za prvé, je vše o použitém softwaru. Dnes můžeme vidět progresivní standardizaci v oblasti, kterou dříve lidé považovali za doplňkovou (tj. dotykové displeje, barevné displeje...) a společnosti musí soutěžit na poli programů a systémů SCADA. Za druhé, integrace s novými „fyzickými“ technologiemi. Tyto trendy, které se snaží podobat běžně používaným zařízením, jsou poslední generací HMI: Snaží se lidem, kteří je používají navodit pocit, že používaní je stejně jednoduché a vizuálně podobné jako u běžných tabletů a chytrých telefonů [13, 14].

2.4.2 HMI v průmyslové automatizaci

Zařízení HMI v zásadě umožňuje vizualizaci a řízení aplikací. Díky prostředkům, jako jsou I/O, SoftPLC, Modbus, Ethercat a operačním systémům (v lepším případě pokud jsou zabudovány), je možná komunikace s jakýmkoliv výrobním systémem.

V závislosti na zařízení se funkce zařízení mohou měnit, pokud jde o připojení, technologii a dokonce i rozměry. Proto můžeme v průmyslové automatizaci najít HMI, které se liší od standardních 4,3 palcových zařízení po sofistikované širokoúhlé displeje s úhlopříčkou 15,6 palců. Pro zjednodušení práce operátorů je k dispozici technologie dotykové obrazovky (odporová nebo kapacitní), která umožňuje intuitivní interakci se stroji a výrobními závody [13].

2.5 Zařízení pro vzdálené řízení – RTU

RTU je zkratka pro Remote Terminal Unit, někdy také nazývaná Remote Telemetry Unit nebo Remote Telecontrol Unit. RTU je elektronické zařízení, které propojuje fyzické objekty do systému SCADA přenosem telemetrických dat do nadřazeného systému. RTU získává různé analogové a digitální parametry pomocí vhodných zařízení pro sběr dat a odesílá data do systému SCADA v požadovaném formátu nebo protokolu.

RTU je ve srovnání s PLC nákladnější a je spíše vhodnější pro vzdálenou správu, oproti tomu PLC implementace je levnější a používá se spíše lokálně. RTU může mít oproti PLC výhodu v jeho programování, PLC vyžaduje znalost specifických programovacích jazyků, oproti tomu RTU lze někdy naprogramovat pomocí jednoduchého webového rozhraní. V jiných případech RTU přichází s instalačním softwarem, který pomůže konfigurovat vstupní toky na výstupní a také pomůže s konfigurací komunikace. Existuje také mnoho RTU, které mají předprogramované moduly, které lze použít pouze pro požadovanou funkci. Některé RTU mohou bát naprogramovány za pomocí jazyků jako Basic, Visual Basic, C++, tyto jazyky vyžadují schopnosti programovat v daném prostředí [15].

2.6 MQQT – MQ Telemetry Transport

MQQT dřívě reprezentující Message Queuing Telemetry Transport a dnes již MQ Telemetry Transport je protokol, který realizuje předávání zpráv mezi klienty prostřednictvím centrálního bodu (brokeru). Protokol byl navržen firmou IBM v roce 1999 ale, dnes je spjatý s firmou Ecplispe foundation a byl standardizovaný konsorciem OASIS (dnes aktuální verze 5.0). Přenos probíhá prostřednictvím protokolu TCP, a jako návrhový vzor je použit formát publisher-subscriber. Existuje právě jeden centrální bod MQTT broker, který zajišťuje výměnu zpráv. Zprávy jsou koncipovány jakožto témata (topic) a zařízení buďto v daném tématu publikuje (publish), což reprezentuje posílání dat do bokeru, který zajišťuje jejich další distribuci nebo je přihlášeno k odběru témat (subscribe). V tom případě boker posílá do zařízení

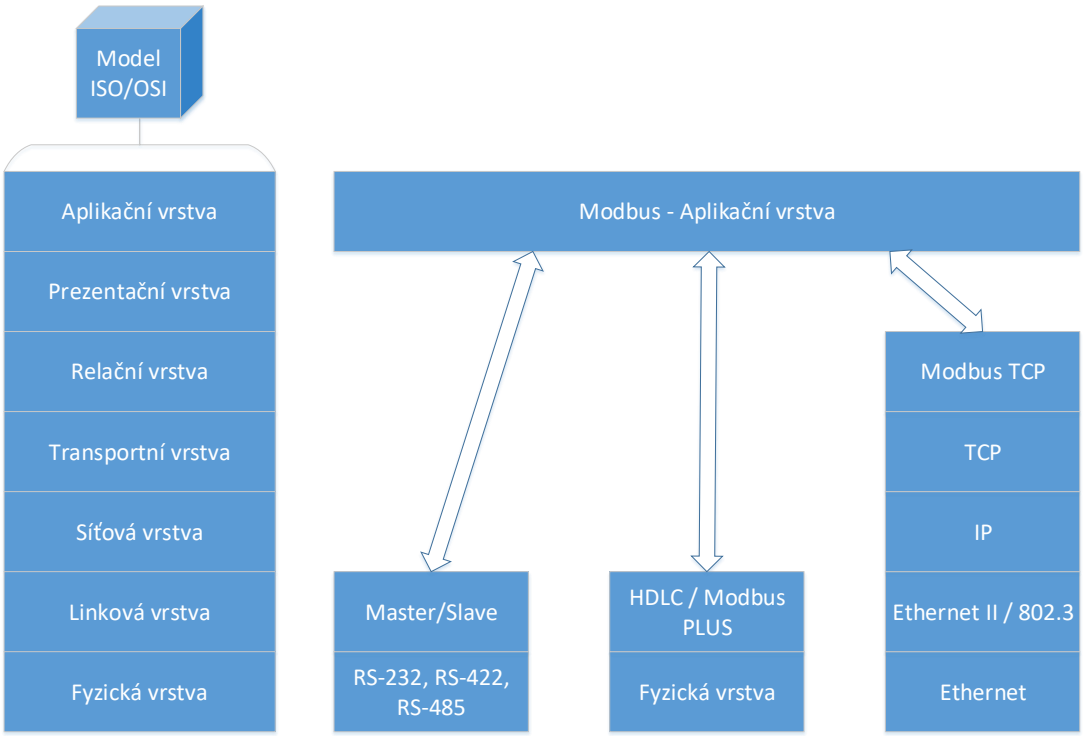
všechny zprávy s daným tématem, ke kterému je zařízení přihlášeno. Zařízení mohou být členy skupin jak subscribe, tak publisher, dle koncepce využití [16].

3 Průmyslové protokoly

V této kapitole budou rozebrány nejčastěji používané protokoly v průmyslu. Bude rozebrána stručně jejich historie a základní komunikační principy včetně zařízení běžně spojena s daným protokolem.

3.1 Modbus

Modbus je sériový komunikační protokol vyvinutý společností Modicon (v dnešní době jedna ze značek koncernu Schneider Electric) publikovaný v roce 1979 pro použití s jeho programovatelnými automaty (PLC). Postupně se z něj stal standard. V jednoduchosti jde o metodu používanou pro přenos informací skrze sériové linky (RS-232, RS-422 a RS-485), rádiové a optické sítě nebo síť Ethernet mezi elektronickými zařízeními viz obrázek 3.1. Zařízení požadující informace se nazývá Modbus Master (nadřízená jednotka) a zařízení, kterým jsou informace dodávány se nazývají Modbus Slave (podřízené jednotky). Ve standardní sítí Modbus je jeden Master a až 247 Slave zařízení, každý s jedinečnou Slave adresou od 1 do 247. Navíc Master má možnost zápisu do zařízení typu Slave.

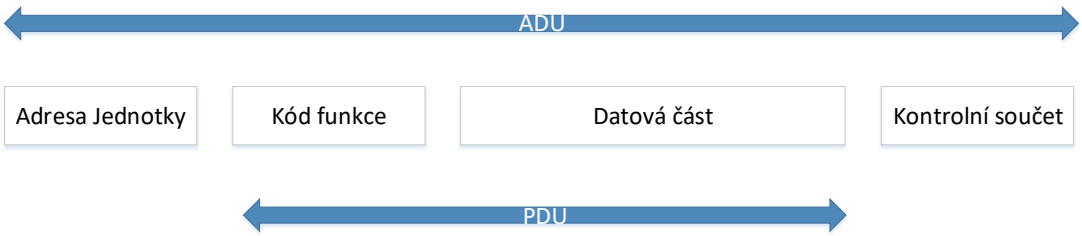


Obr. 3.1: Implementace Modbus v modelu ISO/OSI.

Modbus je open source protokol (otevřený), což znamená, že výrobci mohou tento protokol zabudovat do svých zařízení, aniž by museli platit licenční poplatky. Stal se standardním komunikačním protokolem v průmyslu a je nyní nejčastěji dostupným prostředkem pro připojení průmyslových elektronických zařízení. Je široce využíván napříč všemi průmyslovými odvětvími. Modbus je obvykle využíván k přenosu signálů z přístrojů a řídících zařízení zpět do hlavního kontroléru nebo systému sběru dat, například do systému, který měří vlhkost a teplotu a sděluje výsledky počítači [17]. Modbus je často používán pro připojení dohledového systému se vzdálenou terminálovou jednotkou (RTU) v systémech dohledového řízení a získávání dat (SCADA). Verze protokolu Modbus existují pro sériové linky (Modbus RTU a Modbus ASCII) a pro Ethernet (Modbus TCP/IP).

3.1.1 Popis protokolu Modbus

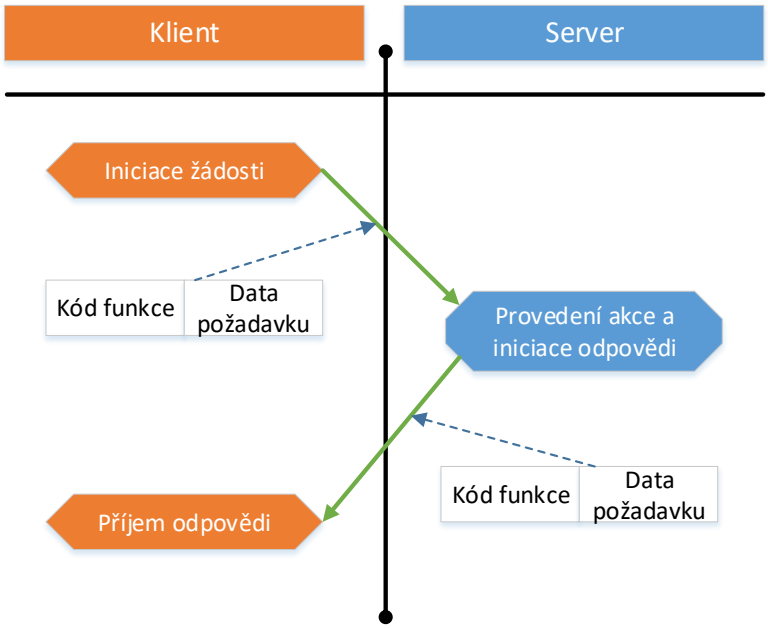
Protokol Modbus je definován jednoduchým PDU (Protocol Data Unit) na úrovni aplikační vrstvy modelu ISO/OSI a v tom případě je tedy nezávislý na komunikaci nižších vrstev ISO/OSI modelu. Při mapování Modbus protokolu na určitou sběrnici nebo síť se mohou přidat další data do ADU (Application Data Unit). PDU je stejný pro komunikaci ve všech typech sítí, ADU je závislý na typu komunikace v dané síti. Základní strukturu rámce protokolu Modbus můžeme vidět na obrázku 3.2.



Obr. 3.2: Obecný rámec protokolu Modbus.

Jednotka PDU je běžnou součástí rámce Modbus, který zahrnuje kód funkce a datovou část. ADU je kompletní rámec a zahrnuje specifickou část pro fyzickou vrstvu. U sériových linek je adresa zařízení předávána v záhlaví ADU a na konci je předáván kontrolní součet (CRC). Maximální velikost ADU v sériových komunikačních linkách je 253 bajtů (1 bajt pro adresy a 2 bajty kontrolního součtu jsou odečteny od maximálních 256 bajtů). V Modbus TCP je maximální délka paketu stanovena na 260 bajtů. Funkce je zakódována do jednoho bajtu a určuje, jakou akci má serverové zařízení provést. Kódy funkcí se pohybují od 1 do 255, přičemž kódy 128 až 255 jsou vyhrazeny pro chybové zprávy ze serveru. Kód 0 se nepoužívá.

Velikost datové části se může lišit od 0 po maximum. Pokud je požadavek zpracován korektně, server vrátí ADU obsahující požadovaná data [18]. Příklad bezchybného provedení požadavku můžeme vidět na obrázku 3.3.

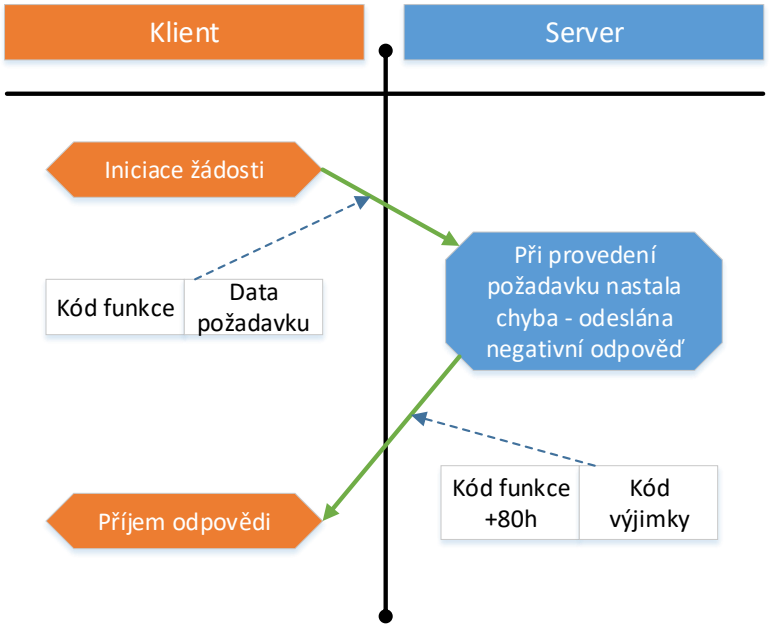


Obr. 3.3: Bezchybná výměna požadavku Modbus.

Pokud dojde k chybě, zařízení vrací kód chyby. V případě normální transakce je funkční kód v odpovědi vrácen beze změny, v případě chyby je místo kódu funkce obsažen kód vyjímky (Exeption Code) pro identifikaci chyby viz obrázek 3.4. Je nutné určit časový limit pro vyčkání na odpověď od zařízení Slave – je zbytečné čekat dlouho na odpověď, která v případě chyby nemusí nikdy dorazit. Jednalo by se o zbytečné vytížení zdrojů a prodlevy při komunikaci.

Protokol Modbus je definován třemi základními typy zpráv (PDU):

1. **Požadavek (Request PDU)** – 1 bajt je *Kód funkce* a n bajtů je *Datová část* požadavku (adresa, proměnné, počet proměnných atd.).
2. **Odpověd (Response PDU)** – 1 bajt *Kód funkce* tj. kopie z požadavku a m bajtů *Datová část* odpovědi (stav zařízení, přečtené vstupy).
3. **Záporná odpověď (Exception Response PDU)** – 1 bajt *Kód funkce + 80h* (neúspěch) a 1 bajt *Kód výjimky* (identifikace chyby).



Obr. 3.4: Výměna požadavku Modbus s chybou.

Kódování dat

V protokolu Modbus je obvyklé kódovat adresy a data ve formátu big-endian. Při přenosu datových položek delších než jeden bajt je nejprve posílán nejvyšší bajt a jako poslední nejnižší bajt. Například při přenosu hexadecimálního čísla 0x1234 zařízení nejprve příjme bajt 0x12 a poté 0x34 [19]. Pro přenos dat jiného typu, např textových řetězců, data a času daného dne atd. může být zvolena jiná metoda kódování.

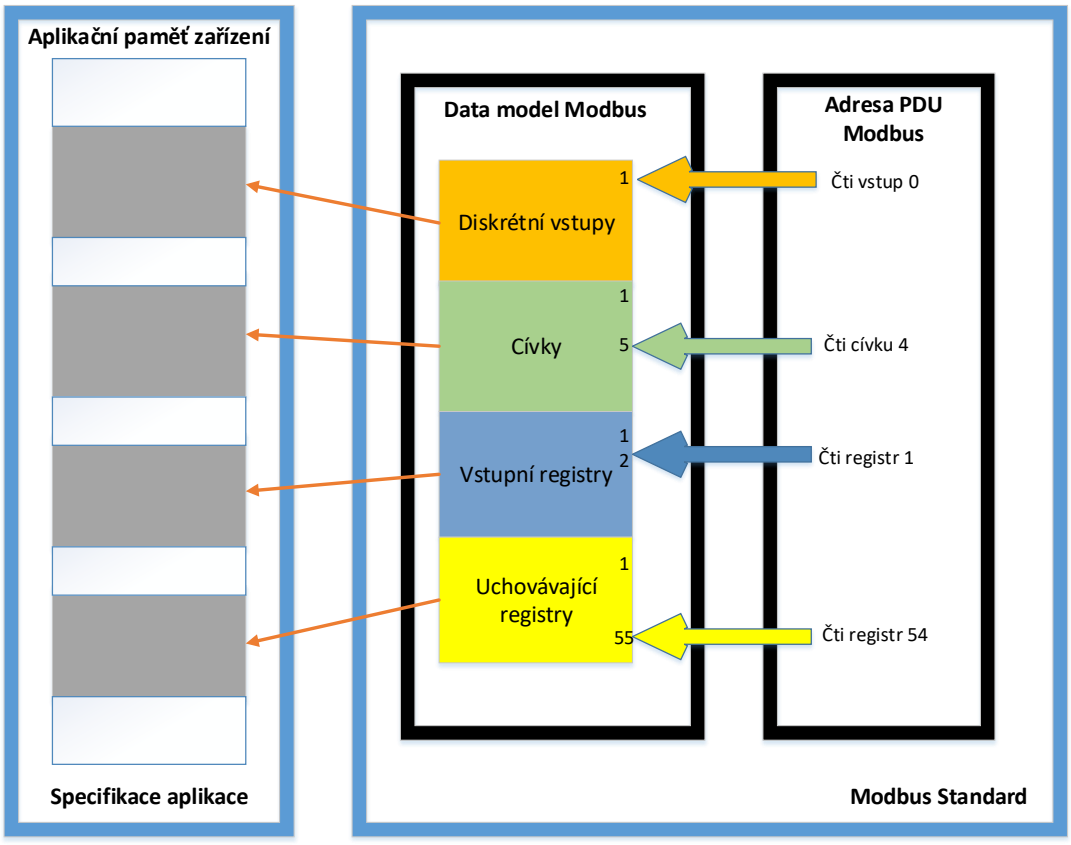
Datový model

Datový model protokolu můžeme rozdělit na sérii oblastí s charakteristickým významem. Definovány jsou čtyři základní oblasti, jak můžeme vidět v tabulce 3.1.

Pokud budeme rozlišovat mezi vstupy a výstupy a bitově adresovanou, slovem adresovanou položkou do adresního prostoru, je chování vždy závislé na konkrétním zařízení. Všechna manipulovaná data musí být uložena v aplikační paměti komunikujícího zařízení. Veškerou primární oblast lze dělit až na 65536 položek, s nimiž se nakládá podle použitého funkčního kódu. Na absolutních adresách obsažených v paměti zařízení nezáleží, protože se pracuje s relativními odkazy. Organizace dat v aplikační paměti komunikujícího zařízení je v rámci jednoho bloku nebo oddělených bloků. Na obrázku 3.5 můžeme vidět datový model s oddělenými bloky a jeho adresování.

Tab. 3.1: Rozdělení oblastí datového modelu.

Primární oblast	Typ objektu	Typ přístupu	Popis
Input Discrete (Diskrétní vstupy)	jeden bit	pouze čtení	data dodaná IO zařízeními
Coils (Cívky)	jeden bit	čtení i zápis	data mohou být měněna aplikačním programem
Input Registers (Vstupní registry)	16-bitové slovo	pouze čtení	data dodaná IO zařízeními
Holding Registers (Uchovávací registry)	16-bitové slovo	pouze čtení	data mohou být měněna aplikačním programem



Obr. 3.5: Datový model Modbus s oddělenými bloky a jeho adresování.

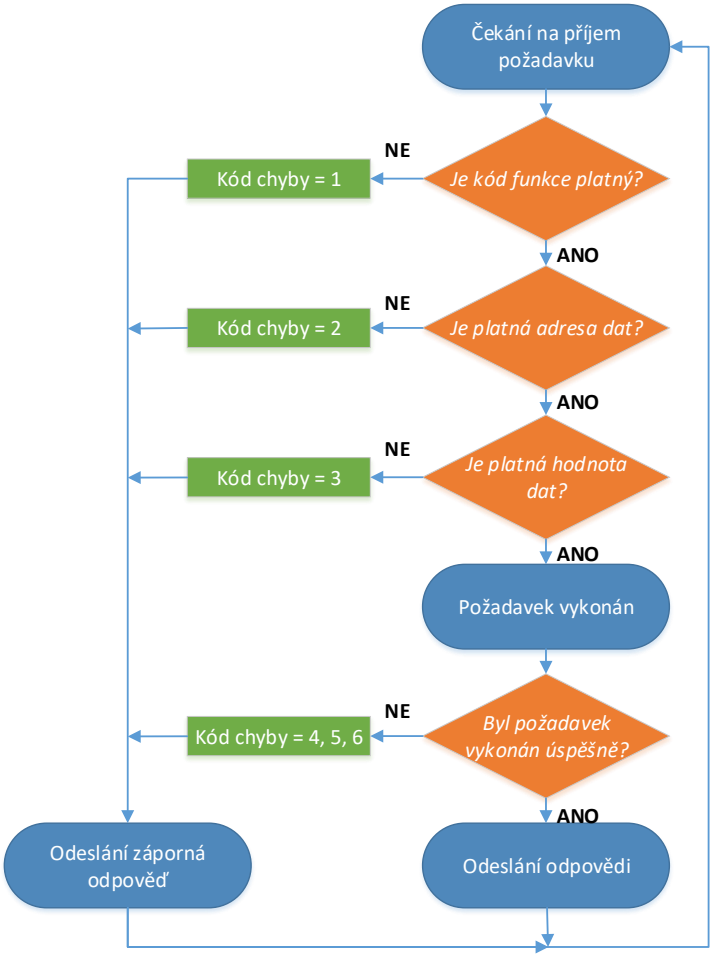
Adresovací model dat

V protokolu Modbus jsou přesně definována adresovací pravidla ve zprávách PDU: **V PDU jsou datové položky adresovány od 0 do 65535.** Dále je definováno

adresování v rámci datového modelu složeného ze 4 datových bloků: **V Modbus datovém modelu jsou položky v datových blocích číslovány od 1 do n.** Na obrázku 3.5 je znázorněn příklad adresování v datovém modelu PDU, pokud bychom vzali požadavek klientské aplikace, je požadavek adresován ve tvaru x-1 (pokud budeme chtít číst diskrétní vstup 1 v datovém modelu PDU, bude čtení reprezentováno 0) [19].

Definování transakce Modbus

Jak můžeme vidět, na obrázku 3.6 je obecný stavový diagram protokolu Modbus, který zobrazuje zpracování požadavku na straně serveru.



Obr. 3.6: Stavový diagram zpracování požadavku na straně serveru.

Jakmile je na straně serveru zpracován požadavek (nezávisle na úspěchu či neúspěchu), je sestavena odpověď a odeslána klientovi. V závislosti na dosaženém vý-

sledku zpracování požadavku je vytvořena jedna ze dvou odpovědí:

1. Response (Pozitivní odpověď) – *kód funkce* v odpovědi se rovná *kódu funkce* v požadavku
2. Exception response (Negativní odpověď) – *kód funkce* v odpovědi se rovná *kódu funkce* požadavku + *80h* – dále je vrácen *kód výjimky*, který udává důvod neúspěchu

Kódy funkcí

Druhým parametrem v každé zprávě Modbus je funkční kód. Funkční kód definuje typ zprávy a typ akce vyžadované podřízeným zařízením (Slave). Parametr obsahuje jeden bajt informací. V Modbus ASCII je kód definován dvěma hexadecimálními znaky, v Modbus RTU je použit jeden bajt. Ne všechna zařízení rozpoznávají stejnou sadu funkčních kódů. V běžném případě při odpovědi podřízeným zařízením je použit stejný funkční kód, jako v požadavku (pokud nenastane chyba).

Kódy funkční lze rozdělit na tři skupiny a to:

1. **Veřejné kódy funkcí** (jasně definované, schválené asociací MODBUS-IDA, unikátní, veřejně zdokumentované, dostupnost testu shody) obsahující čísla 1 až 65 a 111 až 127 a mimo speciálních funkcí definují i akce čtení stavů vstupů, nastavení výstupů, čtení stavů zařízení atd. Kolik funkcí je využito, je závislé na potřebách komunikujícího zařízení a síti, po které je komunikace vedena. Přesný popis funkcí je definován na oficiálních stránkách standardu protokolu Modbus, které jsou dostupné z adresy www.modbus.org [18].
2. **Uživatelsky definované kódy funkcí** (není garantována unikátnost, po projednání s asociací MODBUS-IDA možnost přesunutí mezi veřejné, umožňuje uživateli implementovat funkci) obsahující čísla 65 až 72 a 100 až 110.
3. **Rezervované kódy funkcí** (rezervované kódy funkcí využívané firmami a proto nejsou dostupné pro veřejné použití).

3.1.2 Implementace protokolu Modbus

Ve standardu Modbus je kromě definice aplikační vrstvy v modelu ISO/OSI implementace protokolu na konkrétní typ sběrnice nebo sítě. Takto můžeme rozdělit protokol Modbus na komunikaci po sériové lince a komunikaci na TCP/IP.

Modbus po sériové lince

Modbus protokol po sériové lince je protokol typu Master-Slave je definován na druhé vrstvě (linkové) modelu ISO/OSI. Na první vrstvě (fyzické) modelu ISO/OSI

mohou být použita různorodá sériová rozhraní např. RS-232, RS-485 a jejich další varianty.

Princip protokolu Modbus

Protokol je ve formátu Master/Slave. V jednom okamžiku může být být připojen pouze jeden Master (řídící jednotka) a až 247 Slave (podřízené jednotky). Komunikace je vždy iniciována jednotkou Master, to znamená, že Slave jednotky nikdy nevysílají data bez předchozí žádosti od jednotky Master a navíc Slave jednotky spolu nemůžou navázat komunikaci bez účasti jednotky Master.

Struktura Modbus rámce pro komunikaci po sériové lince je shodná s obecným schématem Modbus rámce ADU. Jakožto kontrola správnosti rámce (Error Check) se využívá metody CRC nebo LRC. Adresa jednotky udává komunikující Slave jednotku a je přenášena v obou směrech komunikace. Linková vrstva modelu ISO/OSI mimo komunikace mezi Master a Slave navíc udává přenosový mód.

Adresovací pravidla protokolu Modbus

Adresní prostor obsahuje 256 různých adres. Adresou 0 je značena Broadcastová adresa, v rozsahu 1–247 jsou individuální adresy Slave jednotek a rozsah 248–255 je rezervován. Master nemá specifickou adresu, pouze jednotky Slave musí mít adresu přiřazenou a to takovou, že v cele síti Modbus je jedinečná.

Režimy vysílání protokolu Modbus

V Modbus protokolu jsou definovány dva režimy vysílání po sériové lince a to Modbus ASCII a Modbus RTU. Režim vysílání určuje v jakém formátu jsou vysílána data, a jak jsou dekódována. Režim RTU podporují všechny jednotky, ASCII režim není povinný, avšak na jedné sběrnici musí všechny jednotky pracovat ve stejném režimu.

Modbus ASCII

Ve vysílacím režimu ASCII je každý 8bitový bajt posílán jakožto dvojice 4bitových znaků ASCII (Amercian Standard Code for Information Interchange). Ve srovnání s režimem RTU je tudiž pomalejší, ale naproti tomu umožuje vysílat znaky s mezerami ve formě 1 sekundy. Pokud není nastaveno jinak, pokud je interval delší než 1 sekunda znamená to, že došlo k chybě. Začátek zprávy je indikován znakem „:“ a konce zprávy dojicí řídících znaků CR a LF. Je možný přenos bez parity a tehdy je paritní bit nahrazen stop bitem [20].

- Formát 10 bitů pro každý bajt v ASCII modu.

- Systém kódování
  Hexadecimálně, ASCII znaky 0–9 a A–F, jeden hexadecimální znak obsahuje 4 bity dat v každém znaku ASCII zprávy. Zařízení monitorují sběrnici nepřetržitě na znak „:“. Když je tento znak přijat, je dekódován další znak dokud nezjistí konec rámce.
- Bity obsažené v bajtu
  1 startovací bit, 7 datových bitů (nejméně významný bit posílán jako první), 1 bit pro sudou/lichou paritu (může být taky bez parity), 1 stop bit pokud je využita parita nebo 2 stop bity bez parity.

Znaky jsou vysílány v pořadí zleva doprava (left ot right) jakožto v pořadí od nejméně významného bitu po nejvíce významný bit. Vysílání bez parity můžeme vidět na obrázku 3.8 a vysílání s paritou můžeme vidět na obrázku 3.9. Formát rámce zprávy ASCII můžeme vidět na obrázku 3.7. Maximální délka Modbus ASCII rámce je 513 bajtů [20].

Start	Adresa	Funkce	Data	LRC	Konec
1 znak :	2 znaky	2 znaky	0 až 2 x 252 znaků	2 znaky	2 znaky CR, LF

Obr. 3.7: Rámec ASCII.

Start	1	2	3	4	5	6	7	Stop	Stop

Obr. 3.8: Bitová sekvence ASCII bez parity.

Start	1	2	3	4	5	6	7	Parita	Stop

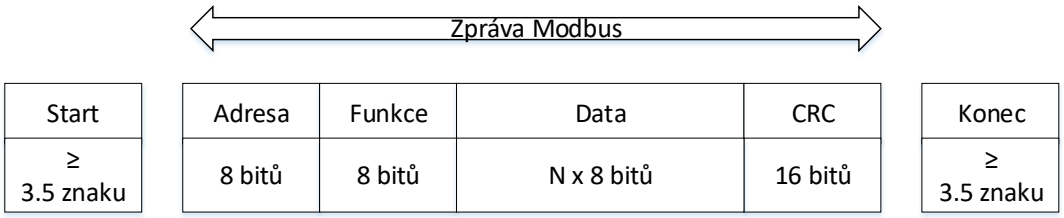
Obr. 3.9: Bitová sekvence ASCII s paritou.

Modbus RTU

V režimu RTU obsahuje každý 8bitový bajt zprávy dva 4bitové hexadecimální znaky. Zpráva se vysílá souvisle, mezera mezi znaky nesmí být delší než 1.5 znaku. Začátek a konec zprávy je definován dle pomlky na sběrnici, která je delší než 3.5 znaku.

- Formát 11 bitů pro každý bajt v RTU modu
- Systém kódování
  8 bit binárně, hexadecimálně 0–9 a A–F. Dva hexadecimální znaky obsažené v každém 8 bitovém poli zprávy.
- Bity obsažené v Bajtu
  1 startovací bit, 8 datových bitů (nejméně významný bit posílán jako první), 1 bit pro sudou/lichou paritu (může být taky bez parity), 1 stop bit pokud je využita parita nebo 2 stop bity bez parity.

Znaky jsou vysílány v pořadí zleva doprava (left to right) jakožto v pořadí od nejméně významného bitu po nejvíce významný bit stejně jako u Modbus ASCII. Vysílání bez parity můžeme vidět na obrázku 3.11 a vysílání s paritou můžeme vidět na obrázku 3.12. Formát rámce zprávy RTU můžeme vidět na obrázku 3.10. Maximální délka Modbus RTU rámce je 256 bajtů [20].



Obr. 3.10: Rámec RTU.

Obr. 3.11: Bitová sekvence RTU bez parity.

Modbus přes TCP/IP

Modbus TCP/IP (neboli také Modbus TCP) je modifikace protokolu Modbus RTU s rozhraním TCP, které běží na Ethernetu. Struktura zprávy Modbus je aplikační

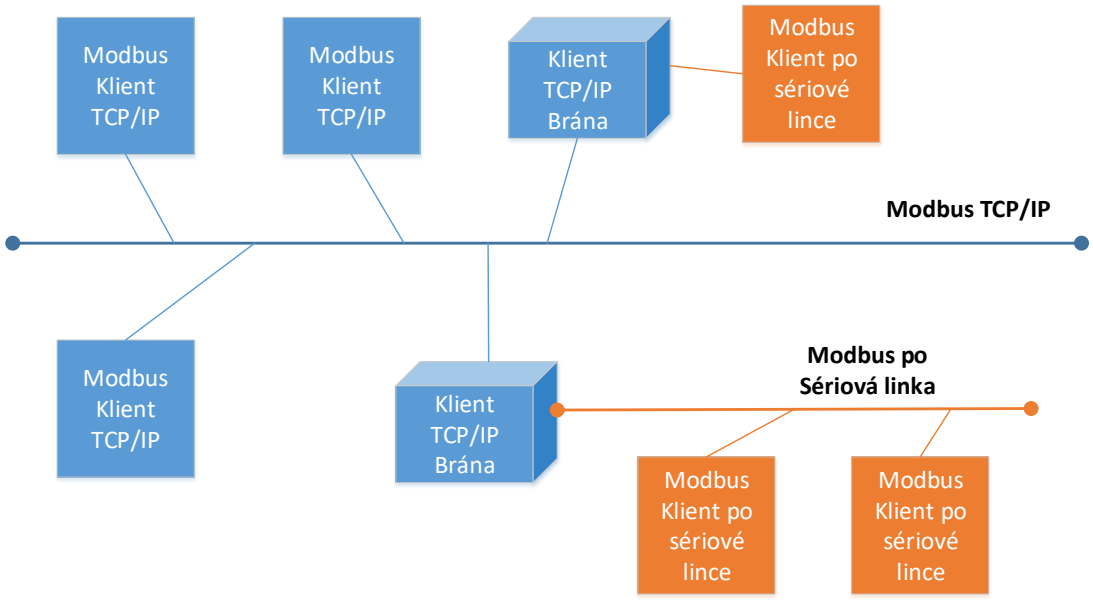
Start	1	2	3	4	5	6	7	8	Parita	Stop

Obr. 3.12: Bitová sekvence RTU s paritou.

protokol, který definuje pravidla pro organizování a interpretací dat nezávisle na médiu pro přenos dat.

Princip protokolu Modbus TCP/IP

Komunikační systém přes Modbus TCP/IP může zahrnovat různé typy zařízení. Zařízení klienta a serveru Modbus TCP/IP připojená k sítí TCP/IP, dále propojovací zařízení, jako je most, směrovač nebo brána, pro propojení sítí TCP/IP a podsítí sériové linky, které umožňují propojení Modbus sériových klientských zařízení a koncových serverů. Modbus TCP/IP komunikační infrastrukturu můžeme vidět na obrázku 3.13.



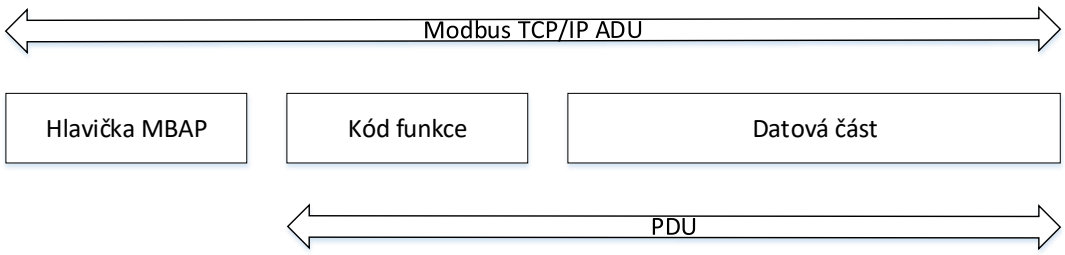
Obr. 3.13: Modbus TCP/IP komunikační infrastruktura.

Tato část bude popisovat zapouzdření požadavku nebo odpovědi protokolu Modbus, když je přenos realizován po síti Modbus TCP/IP. Pro identifikaci aplikační datové jednotky Modbus se v TCP/IP používá vyhrazena hlavička. Její název je

záhlaví MBAP (záhlaví aplikačního protokolu Modbus). Toto záhlaví poskytuje některé rozdíly ve srovnání s aplikační datovou jednotkou Modbus RTU používanou na sériové lince [21].

- Pole „Adresa Slave“ Modbus, které se obvykle používá na sériové lince, je nahrazeno jediným bajtem s názvem „Unit Identifier“ v záhlaví MBAP. Identifikátor jednotky se používá ke komunikaci prostřednictvím zařízení, jako jsou mosty, směrovače a brány, které využívají jednu IP adresu k podpoře několika nezávislých koncových jednotek.
- Všechny požadavky a odpovědi jsou navrženy tak, aby příjemce mohl ověřit, že je zpráva dokončena. Pro funkční kódy, kde má Modbus PDU pevnou délku, stačí stačí pouze funkční kód. Pro funkční kódy nesoucí proměnné množství dat v požadavku nebo odpovědi obsahuje datové pole počet bajtů.
- Když je přenos protokolu Modbus realizován přes TCP, jsou v záhlaví MBAP přenášeny další informace o délce, aby příjemce mohl rozpoznat hranice zprávy, i když byla zpráva rozdělena do více paketů pro přenos. Existence explicitních a implicitních pravidel délky a použití kódu CRC-32 pro kontrolu chyb (na Ethernetu) má za následek nekonečně velkou šanci nezjištěného poškození zprávy s požadavkem nebo odpovědí.

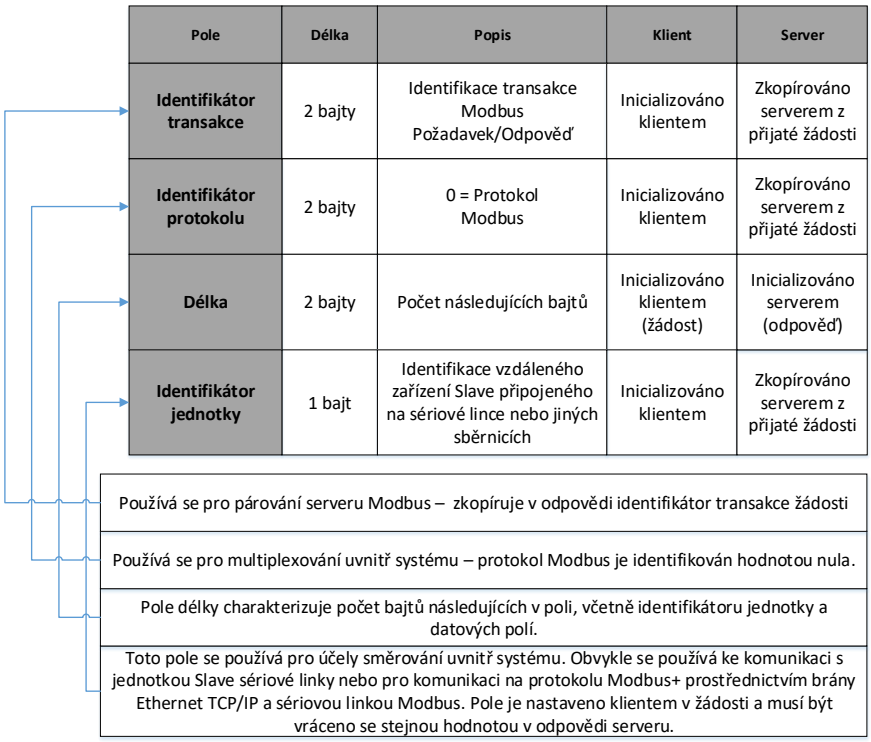
Formát zprávy na protokolu Modbus TCP/IP můžeme vidět na obrázku 3.14. Pokud budeme posílat Modbus TCP/IP ADU, je na TCP vyhrazen registrovaný port s číslem 502 [21].



Obr. 3.14: Modbus TCP/IP zpráva.

Popis MBAP hlavičky

Hlavička MBAP má délku sedmi bajtů a popis jejích polí s komentáři můžeme vidět na obrázku 3.15.



Obr. 3.15: Obsah hlavičky MBAP.

Modbus PLUS

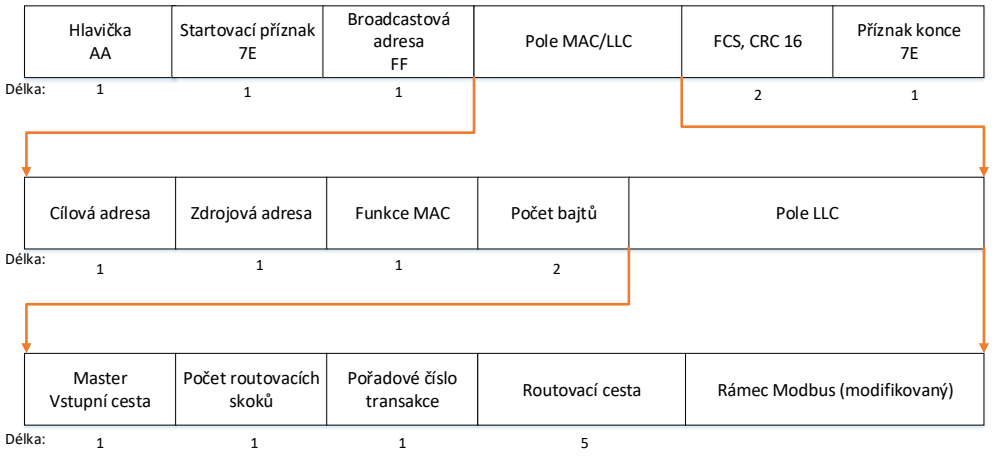
Protokol Modbus PLUS nebo také označován jako Modbus+ je protokol pro linkovou vrstvi modelu ISO/OSI. Protokol byl vyvinut firmou Schneider Automation pro výměnu informací mezi firemními produkty. Protokol je otevřeny a je definován komunikací typu peer-to-peer s výměnou tokenu. V tomto případě se tedy jedná o strukturu sítě Token Ring s fyzickým přístupem na přenosové rychlosti 1 Mb/s.

Každá tato sít podporuje až 64 adresovatelných uzlů neboli zařízení. Pokud budeme uvažovat délku sítě 450 metrů je možno připojit až 32 zařízení, délku sítě můžeme prodloužit za pomocí opakovačů až na 1.8 kilometru. Minimální délka užitého kabelu mezi jednotlivými zařízeními je 3 metry. Jakožto přenosové médium je využívá stíněná kroucená dvojlinka. Sít může být také tvořena za pomocí mostů a zařízení se sériovým rozhraním se mohou připojit za pomocí multipelexorů.

Rozsah jedinečných adres pro každé řídící zařízení v dané síti je 1–64 a je nezávislé na fyzické lokalizaci. Jak již bylo zmíněno dříve, pro připojení většího počtu zařízení je podsítě zapotřebí oddělit mosty. Při komunikaci přijímá zařízení token od předchozího zařízení a odesílá ho následujícímu. Aplikační program tímto získává přístup k registrům všech komunikujících zařízení v dané sítí. Předávání tokenu začíná u zařízení s adresou nejnižší a postupně je předáván zařízením s adresou nejbližší

vyšší. Token se dostává zpátky k zařízení s nejnižší adresou až v momentě, kdy je obsloužené zařízení s nejvyšší adresou. Každé zařízení, které získá token může vyslat zprávy každému zařízení v sítí definicí své adresy a cílové v rámci [22].

Formát zprávy, která je přenášena po síti obsahuje tří hladiny protokolu. Hladiny protokolu jsou HDLC (High-Level Data Link Control) neboli vysokoúrovňové řízení datového spoje, MAC (Media Acess Control), LLC (Logical Link Control) a tyto hladiny můžeme je vidět na obrázku 3.16.



Obr. 3.16: Vzájemné vnořování jednotlivých rámců HDLC, MAC a LLC.

V rámci LLC je již přenášen rámec Modbus, který odpovídá standardnímu PDU. Z rámce ADU je vyjmuta informace o adrese cíle, která se využije na úrovní rámce MAC. Původní CRC v Modbus ADU není využit, protože je kontrola zajištěna v rámci HDLC.

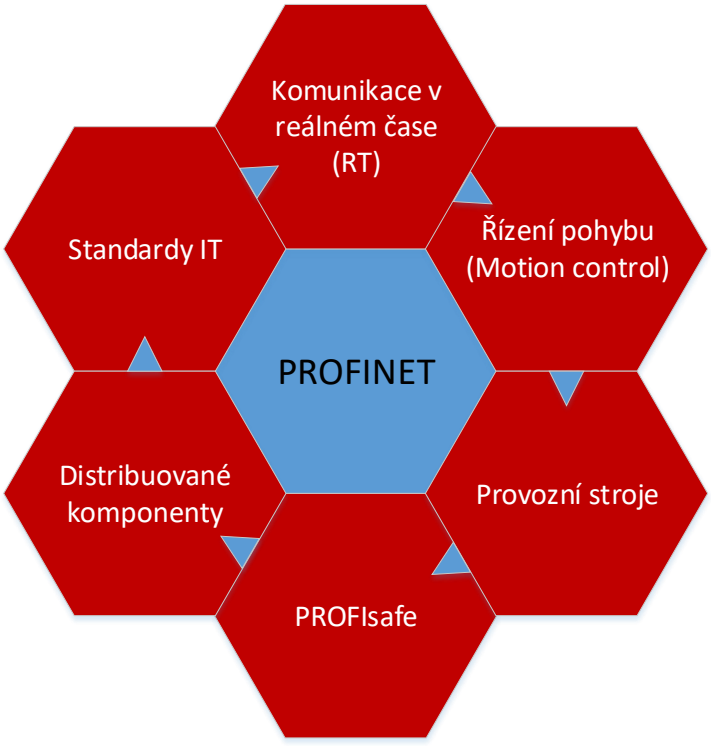
Modbus JBUS

Jak již bylo řečeno Modbus protokol je typu Master/Slave vyvinut firmou MODICON a dále firmou APRIL vyvinut do protokolu s názvem JBUS. Tyto dva protokoly jsou mezi sebou kompatibilní. Výměna dat na obou protokolech funguje stejně, jediný rozdíl je v přístupu k adresám v registrech. V protokolu Modbus je posun nastaven na n+1, kdežto v JSBUSu je přístup přímo n. Počet stanic je rozšířen na 255 a čtení bitů na 2000, čtení slov na 125. Zápis bitů na 1968 a zápis slov na 123. Kódová slova jsou základní společná, ale dle výrobce se ostatní parametry můžou lišit.

3.2 Protokol PROFINET

PROFINET je otevřený a inovativní standard pro průmyslovou automatizaci založenou na průmyslovém Ethernetu, výměna dat mezi zařízeními probíhá na rozdíl od technologie fieldbus po Ethernetu. PROFINET může být použit pro automatizaci výrobních procesů, která vyžaduje menší dobu odezvy než je 100 milisekund. Ostatní aplikace, jako jsou technologie pohonů nebo synchronizace pohybu synchonizované na základě hodin, mohou být doručeny do 1 milisekundy. K bezpečnostním aplikacím může být použit PROFIsafe který je součástí PROFINETu [23].

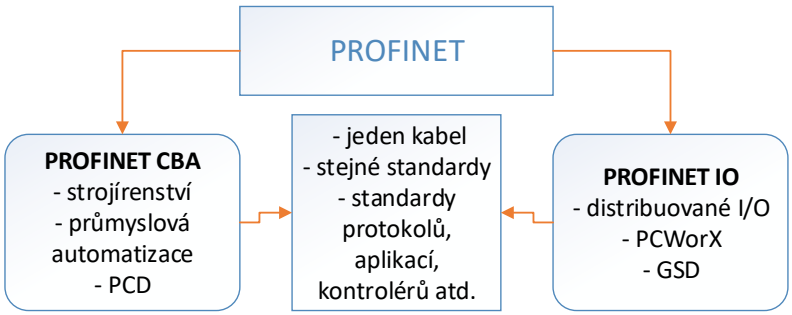
Technologie PROFINET je vyvíjena a publikována společností PROFIBUS/-PROFINET International e.V. (PI) [23]. Doplňuje technologii PROFIBUS, zejména v oblasti rychlosti přenosu dat a implementaci v informačních technologiích (IT). PROFINET využívá standardů informačních technologii jako je TCP/IP a XML pro komunikaci, konfiguraci a diagnostiku zařízení. Jedná se o komplexní komunikační systém, který splňuje veškeré požadavky na průmyslovou automatizaci 3.17.



Obr. 3.17: PROFINET – komplexní standard.

3.2.1 Funkční třídy

PROFINET můžeme rozdělit dvě funkční třídy, které jsou na sobě nezávislé. Jedná se o PROFINET IO a PROFINET CBA (Component Based Automation). PROFINET IO se zabývá ditribuovaných I/O (vstup/výstup) systémů a PROFINET CBA se zabývá distribuovanou automatizací. Rozdělení tříd můžeme vidět na obrázku 3.18.



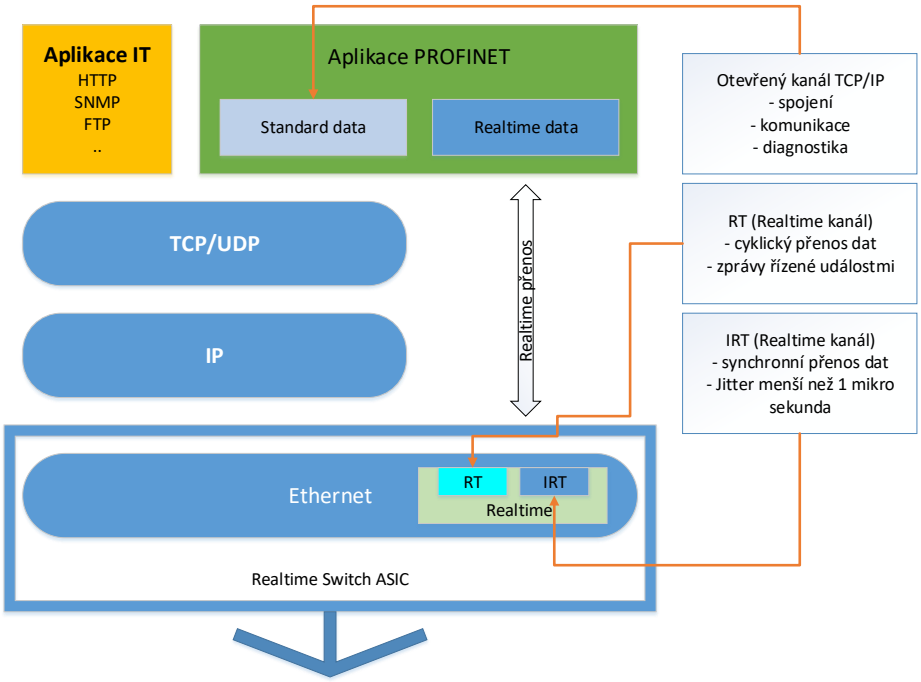
Obr. 3.18: PROFINET – funkční třídy.

- **PROFINET IO:** Pokud vezmeme v potaz PROFIBUS, najdeme mezi těmito dvěma systémy značnou podobnost, například procesní data z průmyslových zařízení se periodicky přenášejí do řídícího systému. PROFINET IO využívá pro výměnu dat s řídícími systémy a dalšími zařízeními tři různé komunikační kanály. Standardní kanál TCP/IP se využívá pro parametrizaci, konfiguraci a acyklické operace čtení a zápisu. NRT (Non-Real Time – přenos, který neprobíhá v reálném čase) je používán pro procesy, které nejsou z hlediska času kritické. RT (Real Time – přenos v reálném čase) se pro standardní cyklický přenos dat a alarmy. IRT (Isochronní Real Time – isochronií přenos v reálném čase) je vysokorychlostní kanál používaný pro aplikace řízení pohybu. Technické vlastnosti zařízení jsou popsány v takzvaném GSD souboru (General Station Description), který je založen na jazyku XML (eXtensible Markup Language) [24].
- **PROFINET CBA:** Tento koncept je určen pro distribuované aplikace průmyslového využití. PROFINET CBA je postaven na standardních technologiích DCOM (Distributed Component Object Model) a RPC (Remote Procedure Call). DCOM je objektově orientovaný mechanismus, který strukturuje způsob, jakým může klient vyhledávat, požadovat a přijímat data ze serveru. DCOM byl vyvinut firmou Microsoft. Objekty DCOM, které jsou odpojeny od PROFINET CBA, se nazývají komponenty. Tyto složky jednají nezávisle a autonomně koordinují své úkony mezi sebou. Zapouzdřené technologické komponenty se

nazývají PROFINET komponenty, které jsou popsány v PCD (PROFINET Component Description). Pro představu mohou vypadat jako černá skříňka s rozhraním venku [24].

3.2.2 Komunikace

PROFINET využívá Ethernet jako komunikační médium. Aby bylo možné Ethernet použít je potřeba implementace protokolů, které jsou definované ve standardu IEEE 803.2. Pro přenos dat jsou implementovány protokoly TCP, UDP nebo IP. Tato implementace však není dostatečná, protože tyto standardy pouze reprezentují základní výměnu dat. Pro funkční komunikaci je potřeba implementace aplikačních protokolů jako např. HTTP, SMTP, FTP, SNMP. Komunikaci v PROFINETu můžeme vidět na obrázku 3.19.

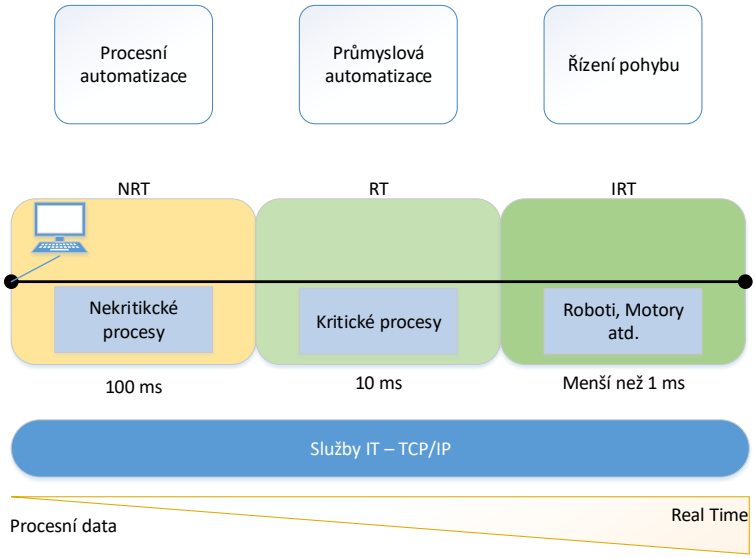


Obr. 3.19: PROFINET – komunikační model.

PROFINET používá tři kanály pro komunikaci s různými výkonnostními třídami dle předpokládaného použití viz obrázek 3.20.

- **NRT (Non-Real Time):** Pro procesy, které nejsou kritické PROFINET používá standardní TCP/IP a UDP/IP pro přenos datových paketů. Časová odezva se pohybuje do 100 ms.

- **RT (Real Time):** Využití pro optimalizovaný výkon výměny dat. Operace čtení a zápisu pro průmyslovou automatizaci vyžaduje převážně vysokorychlostní přenos dat, zatímco standardní TCP/IP nebo UDP/IP tento požadavek uspokojit nemohou. Časová synchronizace spojení se obvykle pohybuje do 10 ms.
- **IRT (Isochronous Real Time):** Využití pro synchronizaci hodin, aplikace pohonů, aplikace pohybů musí být okamžitě uspokojeny. IRT má dobu odezvy kratší než je 1 ms. Této nízké odezvy je dosaženo rozdělením komunikačního cyklu na část, která je deterministická a druhá je otevřená. Deterministickým kanálem jsou přenášeny IRT-telegramy a standardním (otevřeným) jsou přenášeny TCP/IP a RT-telegramy.



Obr. 3.20: PROFINET – Koncept komunikace.

3.2.3 PROFINET IO

Jedná se o komunikační standard, který byl založen mezinárodní organizací PROFIBUS International roku 2004. PROFINET IO je založen stejně jako PROFIBUS CBA na průmyslovém Ethernetu. PROFINET IO navazuje na svého předchůdce PROFIBUS-DP, jedná se o novější standard, který lze lépe implementovat díky využití Ethernetu. Skladba PROFINET IO je tvořena jedním nebo více řídícími členy nazývanými IO-Controller a téměř libovolným počtem podřízených zařízení IO-Device. Je zde také možnost zapojení dozorových zařízení IO-Supervisor. Dále je zde nabízena rekonfigurace systému za běhu a redundantní spojení [24].

Systémový model PROFINETu IO

Zařízení v PROFINETu IO dělíme do tří kategorií dle jejich rolí a funkcí:

- **IO-Controller (řídící člen):**
  IO-Controller bývá většinou součástí PLC a je určen k řízení procesu automatizace. Procesní data mezi podřízeními zařízeními (IO-Device) a řídícím členem (IO-Controller)jsou vyměňována v reálném čase a v cyklických časových intervalech, které jsou předem definovány. PROFINET-IO musí v základu obsahovat alespoň jeden IO-Controller. Při jeho konfiguraci jsou do něj uloženy informace o podřízených jednotkách a dalších členech v sítí. Podporované funkce IO-Controlleru jsou acyklické služby, parametrizace, výměna procesních dat, ovládaní alarmů, přiřazené adresy přes DCP atd. IO-Controller by byl reprezentován v PROFIBUSU zařízením Master.
- **IO-Device (podřízené zařízení):**
  Jedná se o distribuované zařízení, které komunikuje s jedním nebo více IO-Controllerem. Je konfigurováno IO-Controllerem nebo IO-Supervisorem. IO-Device by byl reprezentován v PROFIBUSU zařízením Slave.
- **IO-Supervisor (dozorové zařízení):**
  Většinou se jedná o HMI nebo osobní počítač, který se používá především pro diagnostiku a uvedení IO-Controlleru a IO-Devices do provozu. IO-Supervisor může po nějakou dobu zastupovat funkci IO-Controlleru, připojuje se většinou za běhu systému a to k účelu hledaní a opravy chyb.

Komunikace PROFINETu IO

Pro navázaní komunikace mezi IO-Controllerem a IO-Devices musí být stanoveny komunikační cesty. Ty jsou nastaveny IO-Controllerem během spouštění systému na základě konfiguračních dat. Každá výměna dat je integrována do AR (Application Realtion), v AR stanoví CR (Communication Relations) data explicitně. Výsledkem je, že všechna data pro modelován zařízení, včetně obecných komunikačních parametrů, jsou stažena do IO-Device (může mít více AR vytvořených z různých IO-Controlleru).

Komunikaci můžeme rozdělit na 3 typy:

- **Komunikace Non-real-time:**
  Tato komunikace tvoří přenos dat typu datový záznam, je obsaženo jak čtení tak i zápis datových záznamů a také předpoklad vytvoření spojení, které je tvořeno kontextovým managmentem.

- **Komunikace Real-time Cyklická (RTC):**
  Komunikace RTC slouží k přenosu dat mezi IO-Devices a IO-Controllerem, je konfigurována právě IO-Controllerem. Je monitorován stav CR za pomocí výměny informací o tom v jakém stavu se nachází. RTC vysílá nebo zpracovává data za pomocí stavového automatu. Využívá se pro RT komunikaci uvnitř sítě, mezi sítěmi (RT kanál, UDP protokol) atd.
- **Komunikace Real-time Acyklická (RTA):**
  RTA je schopná přenášet data typu: základní řídící funkce, časová synchronizace, redundantní protokoly, poruchy a diagnostické události. A služby typu: parametrizace IO-Devices, čtení vstupních a výstupních dat atd.

Adresace v PROFINETu

Zařízení v sítí Ethernet vždy komunikují pomocí své jedinečné MAC adresy. V systému PROFINET-IO obdrží každé zařízení symbolické jméno, které jedinečně identifikuje zařízení v tomto systému. Tento název se používá k přiřazení IP adresy k MAC adrese. Využívá se k tomu protokol DCP (Discovery and basic Configuration Protokol). Volitelně je možné využití DHCP protokolu pro přiřazení přiřazení IP adres zařízení.

3.2.4 PROFIsafe v PROFINETu

Protokol PROFIsafe lze použít pro bezpečností aplikace až do kategorie SIL3 (Safety Integrity Level) podle normy IEC 61508 / IEC 62061 nebo PL "e"/ kategorie 4 podle normy ISO 13849. Byl vytvořen pro to aby, bylo zajištěno zabezpečení kritických zařízení a jejích aplikací (vypnutí vypínače, elektrické sítě, zabezpečení dat atd.). PROFIsafe vychází ze stejných principů jako PROFINET a jej možno implementovat i do PROFIBUSU. Komunikace se realizuje zabezpečeně s profilem (využit speciální formát pro uživatelská data a protokol). Ochrané opatření tvořící PROFIsafe jsou – bezpečnostní CRC, komunikace mezi odesilatelem a příjemcem je zaheslována, nové zprávy jsou očekávány s novým pořadovým číslem a potvrzením, bezpečnostní zprávy se číslují sériově [25].

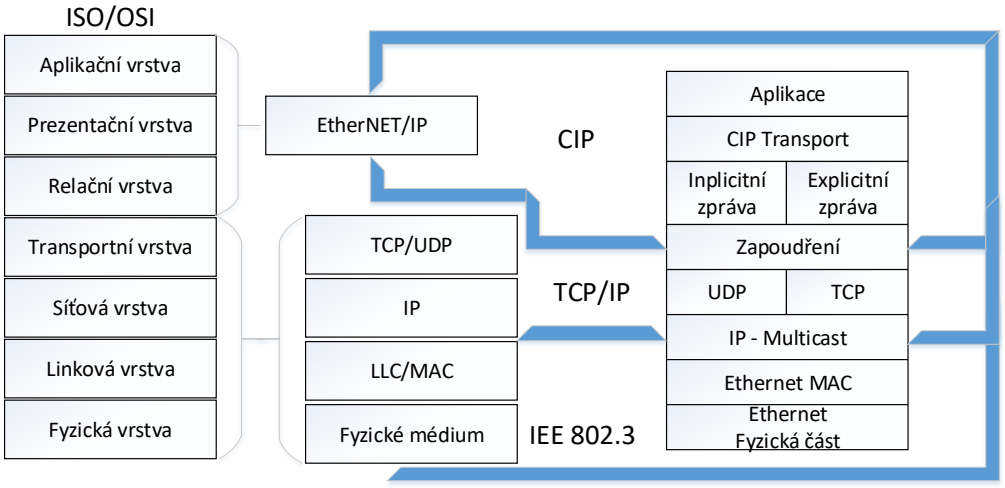
3.3 EtherNET/IP

EtherNET/IP je průmyslový protokol, který byl poprvé představen v březnu 2000 a je výsledkem společného úsilí mezi společnostmi ControlNet International (CI), Open DeviceNet Vendor Association (ODVA) a Industial Ethernet Association (IEA) za účelem vytvoření síťového protokolu, který umožňuje řízení aplikací po klasickém

Ethernetu. Stručně řečeno EtherNET/IP (Ethernet Industrial Protocol) je tradiční Ethernet (standard IEEE 802.3) kombinovaný s protokolem průmyslové aplikační vrstvy zaměřeným na průmyslovou automatizaci. Protokol aplikační vrstvy se nazývá Control and Information Protocol (CIP) a je nezávislý na prvních čtyřech vrstvách modelu ISO/OSI.

3.3.1 Implementace CIP v síti Ethernet

V návaznosti na architekturu Standardu Software/Standard Ethernet využívá Ethernet/IP fyzickou vrstvu, linkovou vrstvu, síťovou vrstvu a transportní vrstvu standardního protokolu Ethernet v kombinací s protokolem CIP přes TCP/IP a UDP viz obrázek 3.21. EtherCAT je jedinečný v tom, že jako jediný průmysloví protokol je založen výhradně na standardech Ethernet. To znamená, že EtherNET/IP používá stejný hardware jako standardní síť Ethernet, takže je lehce dostupný.



Obr. 3.21: EtherNET/IP v modelu ISO/OSI.

Průmyslový protokol CIP

Společný průmyslový protokol (CIP) je síťová aplikační vrstva pro aplikace průmyslové automatizace v reálném čase. CIP definuje strukturu objektu a přenos zpráv, což umožňuje přístup k různým zařízením pomocí společného mechanismu. Každé zařízení v síti EtherNet/IP se prezentuje jako řada datových hodnot nazývaných atributy, které jsou seskupeny do sad objektů. CIP se také používá v zařízeních DeviceNet a ControlNet, takže sdílejí knihovnu objektů a profily zařízení se sítí EtherNet/IP. To zajišťuje kompatibilitu plug-and-play mezi zařízeními od různých

výrobců, stejně jako I/O zasílání zpráv, konfiguraci a diagnostiku v reálném čase v stejné síti bez speciálního softwaru [26].

3.3.2 Komunikace v EtherNET/IP

EtherNET/IP definuje dva typy komunikace a to explicitní, která se používá pro potřebná data, jako jsou informace, a implicitní, která se používá pro data, která je třeba zasílat v reálném čase. Explicitní zprávy jsou přenášeny prostřednictvím protokolu TCP, zatímco implicitní zprávy (které vyžadují vysokou rychlost a nízkou latenci) jsou odesílány přes protokol UDP. Implicitní zprávy mohou používat model producent-spotřebitel. V tomto modelu je zpráva přenášena jednou, bez ohledu na počet zákazníků, a je spotřebována současně mnoha uzly (zařízeními) v síti (technika označovaná jako vícesměrová komunikace). Model výrobce-spotřebitel poskytuje efektivní využití šířky pásma sítě a celkově vyšší rychlosti, zejména když více spotřebitelů potřebuje přistupovat ke stejným datům od výrobce [26].

I přes standardizaci a vysokorychlostní přenos dat EtherNET/IP neodmyslitelně neposkytuje výkon v reálném čase ani provedení záruky v určitém časovém rámci. Důvodem je, že datové pakety TCP/UDP/IP mohou dorazit kdykoli v libovolném pořadí, z jakéhokoli zařízení. EtherNET/IP ve své základní podobě tedy není ideálním řešením pro synchronizované úkoly řízení pohybu. Pro řešení potřeby víceosého distribuovaného řízení pohybu vyvinula organizace ODVA několik možností síťového rozšíření, známých jako CIP-Motion a CIP-Sync (pro synchronizaci hodin mezi osami), které umožňují EtherNET/IP poskytovat deterministický, v reálném čase uzavřený systém při zachování souladu se standardy Ethernetu. Tato rozšíření však zvyšují náklady a složitost jinak jednoduché standardní architektury sítě.

Implicitní a explicitní komunikace

EtherNET/IP podporuje dva typy komunikace. První možností je explicitní zasílání zpráv, kde každá komunikace je samostatným dotazem a odpovědí. Tato komunikace je ze své podstaty pomalejší než implicitní komunikace, protože každý paket vyžaduje režijní informace o tom, co z kterého zařízení je potřeba. Druhou možností je implicitní komunikace tam, kde je spojení mezi zařízeními navázáno v určitém okamžiku a od tohoto okamžiku jsou všechny stanovené informace vyměňovány v nastavených časových intervalech. Každá z těchto komunikací se používá pro různé aplikace a účely. Výběr explicitního nebo implicitního zasílání zpráv často závisí na výběru zařízení, protože každé zařízení může podporovat pouze jeden režim zasílání zpráv.

Explicitní komunikace Explicitní zprávy (klient/server) se nejčastěji používají pro komunikaci v reálném čase, která není časově kritická. V tomto typu zpráv klient (PLC/kontrolér) požaduje informace ze serveru a server odešle požadované informace zpět klientovi. Protože klient požaduje informace prostřednictvím služeb TCP/IP, musí zpráva obsahovat všechny informace, aby server mohl na zprávu explicitně odpovědět. Klient v podstatě řekne serveru, že potřebuje tyto konkrétní informace, s tímto specifikovaným formátováním a žádá o jejich zaslání. Server poté odpoví správně naformátovanou zprávou s požadovanými informacemi. Tato schopnost konfigurace a monitorování funguje dobře pro zasílání zpráv v reálném čase, protože klient může kdykoli odeslat požadavek na zprávu a server může reagovat kdykoliv, když je k dispozici [27].

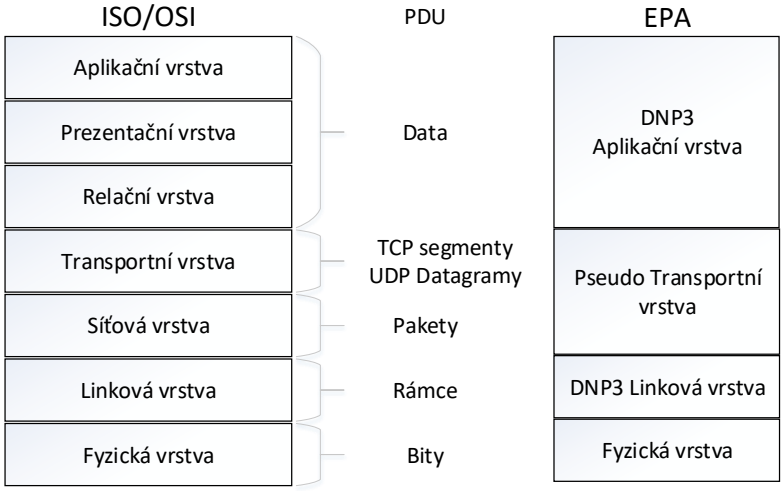
Implicitní komunikace Implicitní zasílání zpráv (I/O Messaging) se používá pro časově kritické aplikace, jako je synchronizace nebo řízení pohybu v reálném čase. Implicitní zprávy se nazývají I/O zprávy, které se často používají pro vzdálené I/O aplikace. Tato komunikace je mnohem efektivnější než explicitní zasílání zpráv, protože Master a Slave jsou předem nakonfigurovány tak, aby přesně tzn. implicitně věděly, co od této komunikace očekávat. Implicitní zprávy v podstatě zkopírují do zprávy nastavené množství dat s minimem dalších informací. Stanici Master a Slave není třeba předávat spoustu informací, protože oba vědí, co ve zprávě očekávat a co poslat zpět. Význam dat je implikovaný. Takže nastavení implicitních zpráv je jednoduché a rychlé. Master potřebuje pouze nastavení, aby věděl, jaká data by měl přijímat a odesílat a ke kterému zařízení EtherNET/IP se musí připojit. Poté jsou data přenášena rychlostí, která je předem určena, obvykle v rozsahu 5 až 20 milisekund [27].

3.4 Protokol DNP3

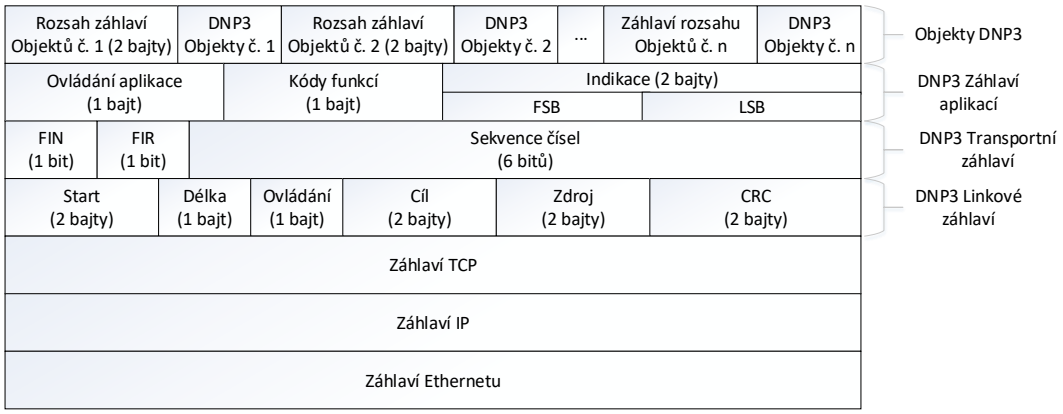
DNP3 neboli Distributed Network Protocol (Distribuovaný síťový protokol) představuje sadu protokolů pro komunikaci, které jsou implementovány mezi komponentami v systémech automatizace procesů. Nejčastější využití zastávají ve vodárenském průmyslu a elektrických sítích. Byl vyvinut pro komunikaci mezi různými typy zařízení pro sběr a kontrolu dat. V systémech SCADA využívá protokolu DNP3 nadřazená řídí stanice SCADA (např. řídí centra), vzdálené terminálové jednotky (RTU) a inteligentní elektronická zařízení (IED) [28].

3.4.1 Struktura protokolu DNP3

Protokol DNP3 se skládá ze tří hlavních vrstev, a to linkové vrstvy, pseudo transportní vrstvy a aplikační vrstvy a může být využíván na sériové sběrnici nebo v sítí TCP/IP. Oproti sedmivrstvému modelu ISO/OSI byl počet vrstev snížen ze sedmi na tři, tento model se nazývá Enhanced Performance Architecture (EPA) a jeho srovnání s modelem ISO/OSI můžeme vidět na obrázku 3.22. V případě využití TCP/IP jsou zprávy protokolu, které obsahují všechny vrstvy posílány prostřednictvím protokolu transportní vrstvy TCP [28]. Grafické znázornění DNP3 pro přenos TCP/IP můžeme vidět na obrázku 3.23.



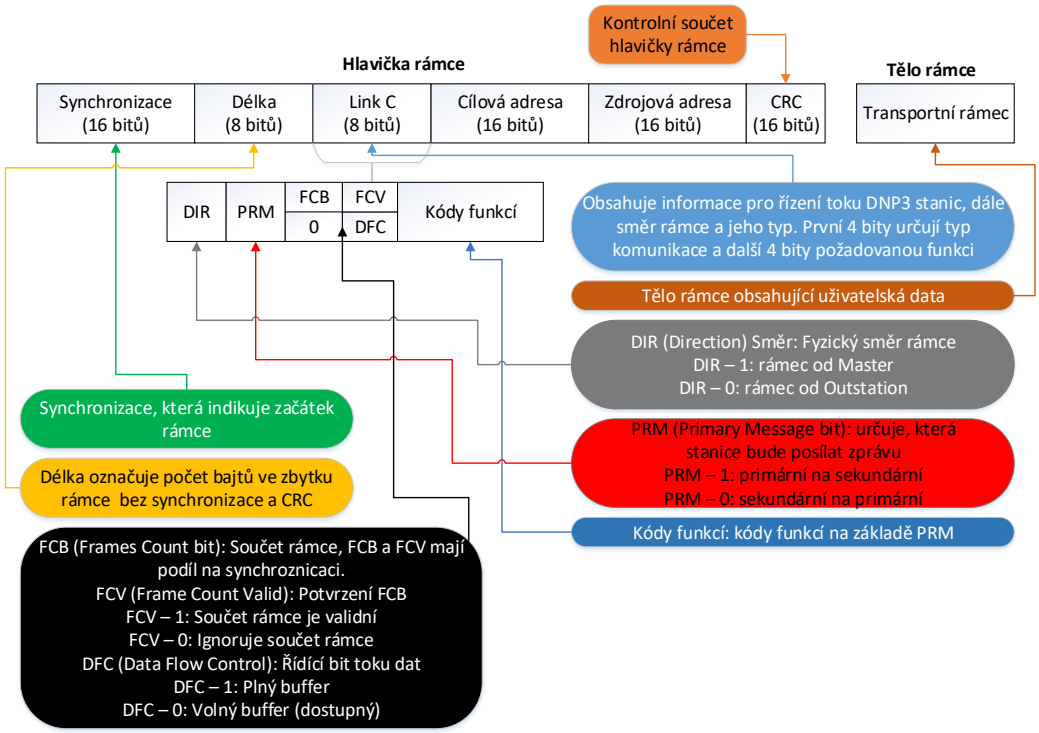
Obr. 3.22: Srovnání modelu EPA s ISO/OSI.



Obr. 3.23: DNP3 přes TCP/IP.

Linková vrstva protokolu DNP3

Linková vrstva má na starost zajištění spolehlivosti přenosu, dělá to tak, že poskytuje detekci chyb a hlídá možnost výskytu duplicitních rámců. Linková vrstva pracuje s rámci, DNP využívá formát rámce FT3 což je FrameTyp3. Pole v hlavičce rámce pro cílovou a zdrojovou adresu má velikost 16 bitů. Spolehlivost přenosu je zajištěna několika kontrolními součty (CRC), které se počítají pro každých 16 bajtů a jeden CRC vždy pro hlavičku rámce. Na potvrzení přijatých dat se využívá zpráva o fixní dálce 10 bajtů [28]. Strukturu všeobecného DNP3 rámce můžeme vidět na obrázku 3.24.

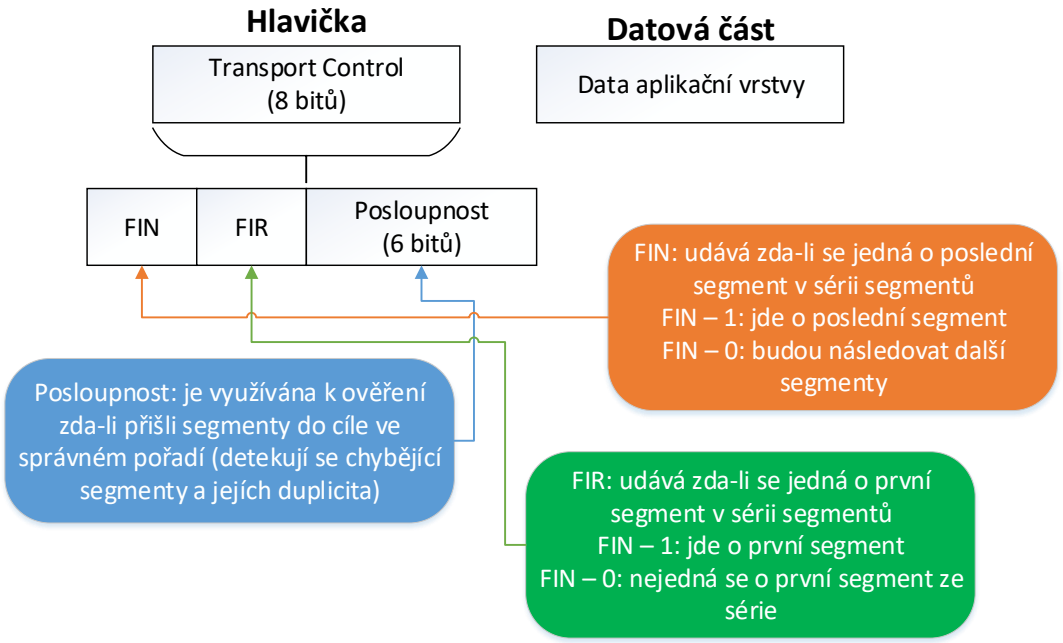


Obr. 3.24: Struktura Linkového rámce.

Pseudo transportní vrstva protokolu DNP3

Pseduo transportní vrstva se nachází v modelu EPA mezi linkovou a aplikační vrstvou, je tvořena hlavičkou a daty aplikační vrstvy. Pseudo transportní vrstva se jí říká proto, že nedochází k adresování a ani není obsažen mechanizmus na potvrzování zpráv nebo zajištění spolehlivosti. Jedná se o nekompletní transportní vrstvu, jak ji známe z modelu ISO/OSI, a proto má v názvu pseudo. Transportní hlavičku obsahují pouze rámce, které obsahují data aplikační vrstvy. Využití této vrstvy bylo ospravedlněno z důvodu dělení dat aplikační vrstvy na menší segmenty, které jsou mnohem

vhodnější pro přenos na linkách s vysokým vytížením [28]. Strukturu pseudo transportní vrstvy můžeme vidět na obrázku 3.25.



Obr. 3.25: Struktura pseudo transportní vrstvy.

Aplikační vrstva protokolu DNP3

Aplikační vrstva odpovídá za operace, které jsou definovány zařízením. Rozděluje data aplikační vrstvy na fragmenty. Maximální velikost je závislá na vyrovnávací paměti přejímacího zařízení. Normální rozsah je 2048 až 4096 bajtů. Zpráva, které je větší než jeden fragment, vyžaduje více fragmentů. Fragment o velikosti 2048 bajtů musí být pseudo transportní vrstvou rozdělen do 9 rámců a pokud je velikost 4096 bajtů tak do 17 rámců [28].

Akronymy použité pro oddělení řídících dat od přenášených dat:

- **APDU (Application Protocol Data Unit:** Jedná se aplikační datovou jednotkou, která je složena z APCI a ASDU.
- **APCI (Application Protocol Control Information:** Záhlaví aplikační jednotky, které určuje její délku, typ, posloupnosti atd.
- **ASDU (Application Service Data Unit:** Příkazy, zasílané mezi funkcemi řídícího a řízeného systému.

3.4.2 Statická data a data událostí

Aplikační vrstva spolupracuje s pseudo transportní vrstvou a linkovou vrstvou, což umožňuje spolehlivou komunikaci. Jsou poskytovány standardizované funkce a formátování dat, se kterými může uživatelská vrstva spolupracovat. V protokolu DNP3 se termín statický používá ve vazbě na data a odkazuje na skutečnou hodnotu. Statická binární data se tedy vztahují k současnému stavu zapnuto nebo vypnuto. Statická analogová vstupní data obsahují hodnotu analogového signálu v okamžiku jeho přenosu. Jednou z možností, je vyžádat si některá nebo všechna statická data v zařízení outstation.

Protokol DNP3 sdružuje důležité události. Například se může jednat o změny stavu, hodnoty překračující určitou prahovou hodnotu, přechodná data a nově dostupné informace. Událost nastane, když se binární vstup změní ze stavu zapnuto na vypnuto nebo když se analogová hodnota změní o více než je nakonfigurovaný limit. DNP3 poskytuje možnost hlášení událostí s časovými razítky a bez nich, takže v případě potřeby bude mít stanic Master informace k vygenerování zprávy o časové posloupnosti.

Uživatelská vrstva na stanici Master může nasměrovat DNP3 na vyžádání událostí. Ve většině případech je Master aktualizován rychleji, pokud tráví většinu času dotazováním událostí z outstationu a pouze občas požaduje statická data.

Události v protokolu DNP3 jsou klasifikovány do tří tříd. Při vzniku DNP3, byly události třídy 1 požadovány za události s vyšší prioritou než události třídy 2 a třída 2 měla vyšší prioritu než třída 3. Dnes je možnost konfigurace priority každé třídy v závislosti na použité aplikaci.

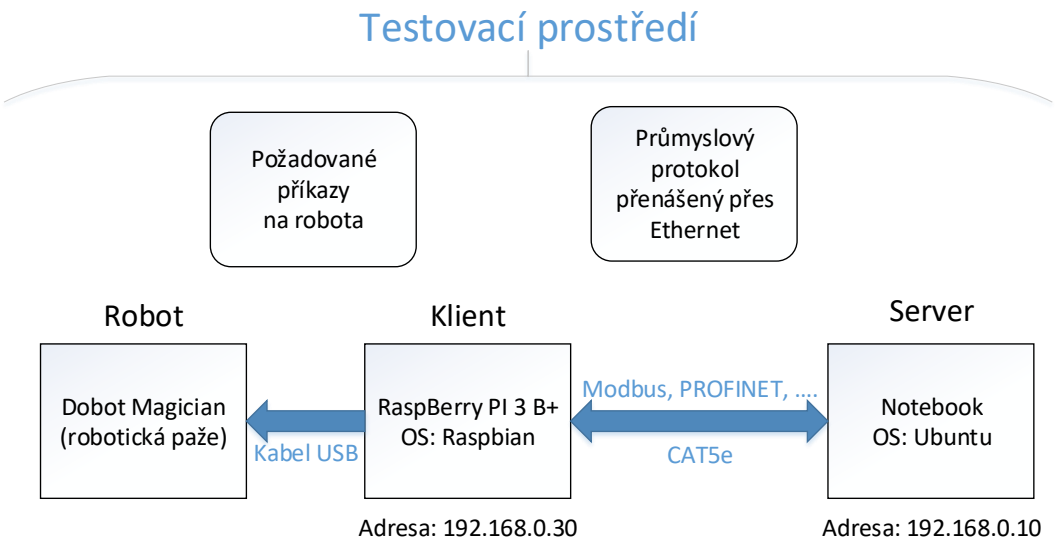
3.4.3 Integrace DNP3 do TCP/IP

Záměr při tvoření protokolu DNP3 byl původně přenos po sériové lince, ale v dnešní době s rozmachem Ethernetu, bylo potřeba jej implementovat do tohoto světa. DNP3 byl vnořen mezi protokoly modelu ISO/OSI a to tak, že rámce na linkové vrstvě přecházejí v Ethernetu jako segmenty TCP nebo UDP datagramy. Pokud by byla

provedena analýza paketu DNP3 bychom viděli, že výše zmíněné tři vrstvy protokolu DNP3 jsou zapouzdřeny jako data aplikační vrstvy modelu ISO/OSI.

4 Implementace testovacího prostředí

Testovací prostředí bylo navrženo tak, aby mohla být otestována funkcionalita průmyslových protokolů pro komunikaci s průmyslovým robotem. Prostředí obsahuje řídící jednotku – v našem případě server, který je zastoupen notebookem (zde může být využita i pevná stanice nebo některé průmyslové řešení). Jako klient (middleware vrstva) je využit miniaturní počítač Raspberry Pi 3 Model B+ a jako robot je využita robotická paže Dobot Magican Robotic Arm. Server je s klientem propojen za pomocí UTP kabelu kategorie CAT5e po kterém je komunikace vedena průmyslovým protokolem (instrukce pro Dobota jsou zapouzdřeny uvnitř průmyslového protokolu). Klient je s robotickou rukou propojen přes rozhraní USB. Schéma zapojení můžeme vidět na obrázku 4.1. Detailnějším rozborem jednotlivých prvků a způsobu komunikace budou věnovány další podkapitoly.



Obr. 4.1: Schéma zapojení testovacího prostředí.

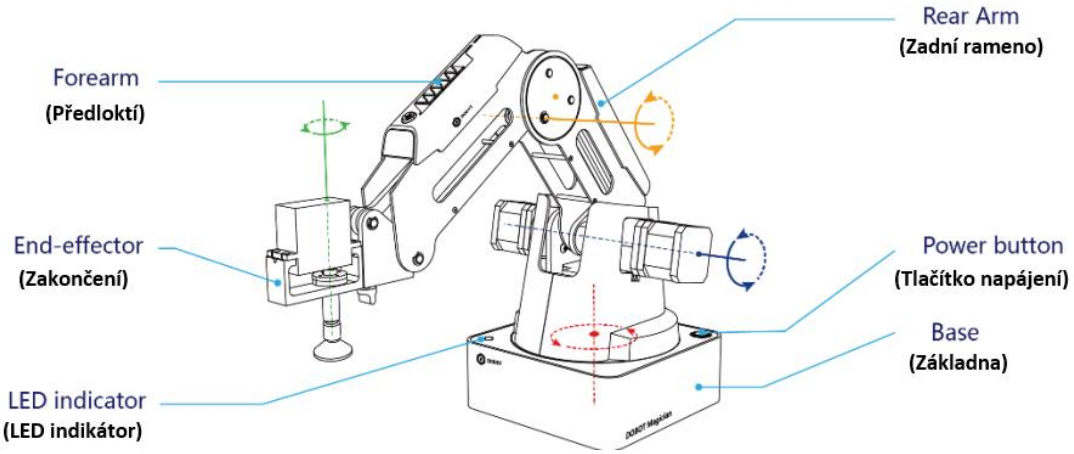
4.1 Dobot Magician

Dobot Magican je multifunkční robotická paže v kompaktní velikosti pro výzkumné a testovací účely. Začínala jako startup, ale výběr peněz pro tvorbu konečného řešení několikanásobně překročil požadovanou částku pro začátek výroby. Dokáže realizovat spoustu zajímavých funkcí jako je 3D tisk, laserové gravírování, psaní a kreslení. Pro veškeré základní operace je dodáván software Magician Studio, kde je možné psát skripty, tvořit bloková schémata, ovládat Dobota za pomocí myši a spoustou

dalších možných funkcionalit. Důležité je, že Dobot podporuje také sekundární vývoj za pomocí rozšiřitelných I/O rozhraní, což zvyšuje jeho možnost nasazení závislé pouze na kreativitě jedince.

4.1.1 Konstrukce Dobota a pracovní prostředí

Dobot je tvořen základní stanicí ve které je ukotven, v základně je rovněž obsažena řídící jednotka DfRduino Mega2560 V3 (srovnatelná s Arduino Mega 2560) společně s množstvím použitelných rozhraní. Dále je tvořen základní paží, předloktím a zakončením (end-effector), na nějž lze aplikovat spoustu doplňků dle aplikace použití. Části robotické paže jsou propojeny servomotory, které tvoří klouby. Vzhled robotické paže Dobot můžeme vidět na obrázku 4.2.

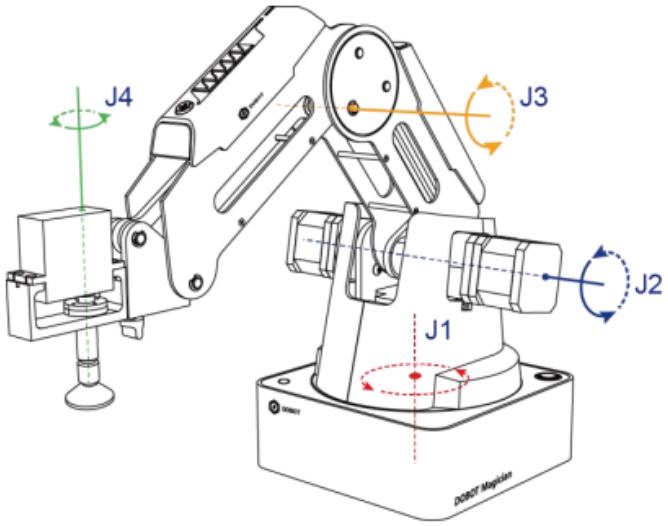


Obr. 4.2: Vzhled robotické paže Dobot Magician [30].

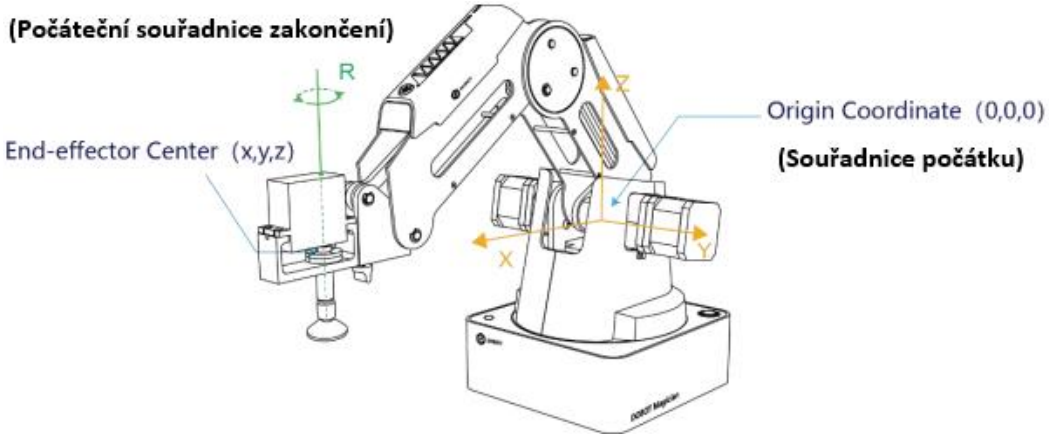
4.1.2 Souřadnicový systém

Robotická paže Dobot využívá dvou souřadnicových systémů a to kloubový (Joints) a kartézský souřadnicový systém. Znázornění těchto systémů můžeme vidět na obrázku 4.3 pro kloubový a obrázku 4.4 pro kartézský.

- **Kloubový souřadnicový systém:** Souřadnice systému jsou určeny pohybovými klouby. Pokud není nainstalováno zakončovací zařízení obsahuje Dobot tři klouby a to J1 (Joint1 – Kloub1), J2 (Joint2 – Kloub2) a J3 (Joint3 – Kloub3), což jsou všechno rotující klouby poháněné servomotory. Pokud je Dobot vybaven zakončovacím zařízením jako je např. přísavka poháněna vakuovým čerpadlem nebo sadou pro uchycení předmětu, je přidán další kloub



Obr. 4.3: Kloubový souřadnicoví systém [30].



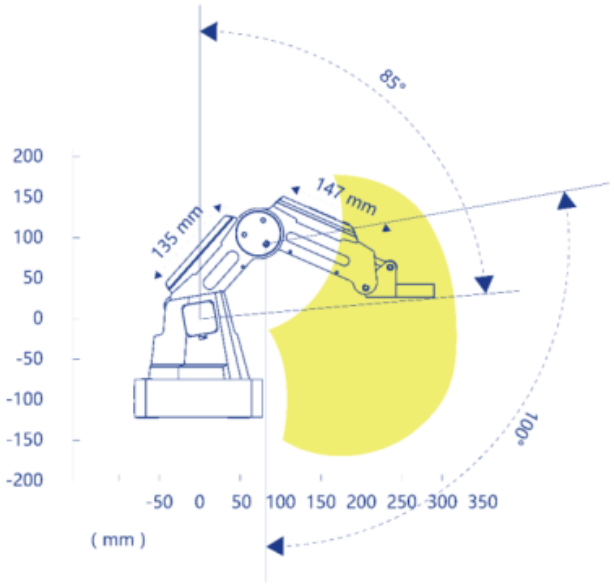
Obr. 4.4: Kartézský souřadnicoví systém [30].

J4 (Joint4 – Kloub4). Pozitivní směr pohybu kloubu je proti směru hodinových ručiček a negativní směr pohybu je po směru hodinových ručiček [30].

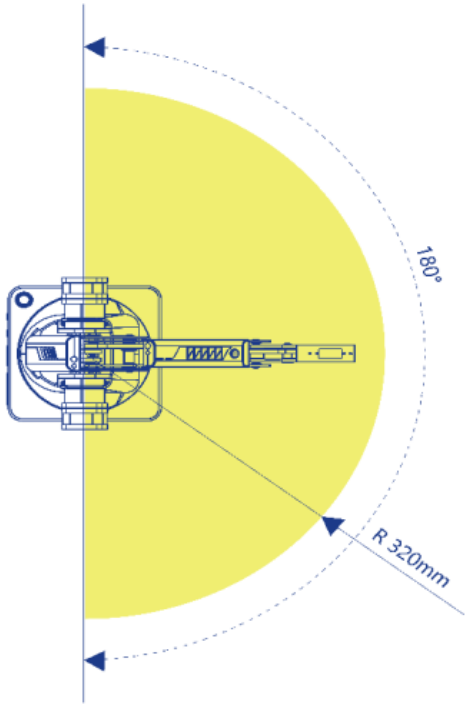
- **Kartézský souřadnicový systém:** Souřadnice kartézského souřadnicového systému jsou určeny umístěním základny. Počátek je určen středem tří servomotorů (základna, předloktí a rameno). Osa X je kolmá k základně a definuje pohyb vpřed a vzad. Osa Y určuje rotační pohyb okolo základnové stanice a osa Z definuje zdvih ramene oproti základně. Osa R je poloha středu servomotoru vzhledem k počátku robotického ramene, jehož kladný směr je stejně jako u kloubů proti směru hodinových ručiček. Osa R je nezávislá na pohybu

kloubů, ale otáčí pouze koncové zařízení [30].

Maximální rozsah pohybu paže Dobot můžeme vidět na obrázcích 4.5 a 4.6.



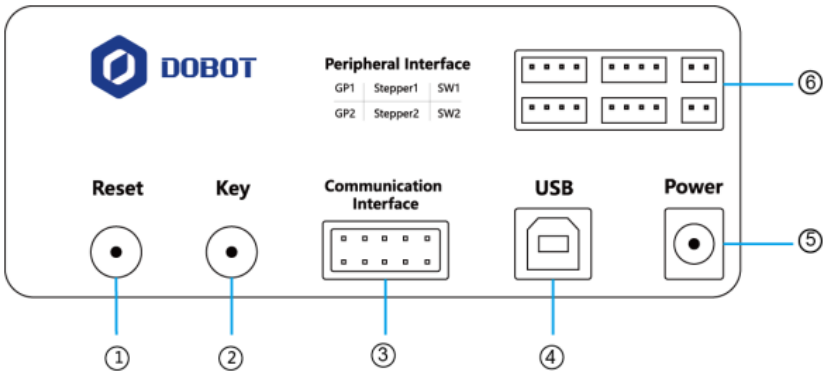
Obr. 4.5: Pracovní prostor Dobota pohled z boku [30].



Obr. 4.6: Pracovní prostor Dobota pohled z vrchu [30].

4.1.3 Popis komunikačních rozhraní

Komunikační rozhraní robotické paže Dobot Magician jsou umístěna na zadní straně základní stanice a také na předloktí paže. Na obrázku 4.7 je zobrazena základní stanice s komunikačními rozhraními [31].



Obr. 4.7: Komunikační rozhraní na základně Dobota [31].

- **1:** Tlačítko Reset, slouží k resetování programu. Během resetování se kontrolka LED na základně rozsvítí žlutě. Doba trvání resetu je cca 5 sekund, pokud se indikátor LED rozsvítí zeleně tak resetovaní proběhlo v pořádku.
- **2:** Tlačítko funkčního klíče slouží k spuštění offline programu (krátké stisknutí), pokud je podrženo po dobu 2 sekund, tak se Dobot nastaví do výchozí pozice.
- **3:** Rozhraní I/O, UART, slouží pro připojení modulů pro komunikaci Dobota přes Blutetooth (BT) nebo bezdrátově přes Wi-Fi.
- **4:** Rozhraní USB pro připojení Dobota k počítači. V našem případě pro připojení Dobota ke klientovi (Raspberry Pi 3 Model B+).
- **5:** Rozhraní určené pro napájení Dobota, zde je připojen napájecí adaptér.
- **6:** Do tohoto rozhraní se připojují periferní zařízení jakožto vakuová pumpa pro přísavku, periferní senzory atd.

4.1.4 Dostupné knihovny

Pro robotickou paži Dobot Magician je dostupná oficiální knihovna **libdobot** [32], což je Linuxová knihovna zkompilovaná pro I306:x64-x86, je psaná v QT (Platforma pro vývoj aplikací pro stolní PC, mobilní zařízení atd. a mezi podporované platformy

patří Linux, Mac OS, Windows, iOS, Android...) což není programovací jazyk sám o sobě, jedná se o objekty psané v programovacím jazyce C++. Bohužel v závislosti na našem testovacím prostředí, kdy klient je tvořen Raspberry Pi 3 Model B+, který funguje na architektuře ARM, není možno knihovnu bez opětovné kompilace pro architekturu ARM použít.

Jako vhodná alternativa k použití se ukázala knihovna **pydobot** v aktuální verzi 1.0.2 [33], která je psaná v jazyce Python (nezávislá na architektuře, potřebuje pouze Python interpreter). Bohužel, na rozdíl od oficiální knihovny **libdobot**, neobsahuje všechny možné dostupné metody pro ovládání Dobota (jeho periferií), chybí například ovládání osy L, která je tvořena posuvnou kolejnicí po které se může Dobot pohybovat. Knihovna je dostupná přes standardizovaný balíčkový systém pip (standardní systém správy balíčků používaný k instalaci a spravě softwarových balíčků napsaných v Pythonu).

Pro komunikaci s Dobotem bude tedy z předchozích zjištění využívána knihovna **pydobot** a to z důvodu využití Raspberry Pi 3 Model B+, pro kterou je oficiální knihovna libdobot.so nepoužitelná a protože v operačním systému využívající Raspberry je již ze základu nainstalován Python interpreter.

4.2 Volba průmyslových protokolů

V závislosti na charakteru testovacího prostředí byly vybrány protokoly, které mohou komunikovat po Ethernetu, kdy propojení serveru (Notebook) a klienta (Raspberry) je realizováno přes Ethernetový port (konektor RJ-45) kabelem kategorie CAT5e.

4.2.1 Modbus TCP

Pro Modbus TCP vzhledem k jeho velikému rozšíření existuje veliký sortiment dostupných knihoven. Pro následnou implementaci byla vybrána knihovna s názvem **pymodbus** ve verzi 2.3.0 [34], která je dostupná přes pip. Byla vybrána, protože se jedná o Python knihovnu a náš klient obsahuje Python interpreter a protože dobře zapouzdří přenos složitějších datových struktur, jako je např. string. Také je ke knihovně vytvořena kvalitní dokumentace a proto byla vhodná pro implementaci.

4.2.2 EtherNET/IP

EtherNET/IP není tak moc rozšířen oproti Modbusu TCP a byla nalezena pouze jedna vhodná knihovna. Vzhledem k využití Python interpreteru byla vybrána knihovna s názvem **cpppo** v aktuální verzi 4.0.6 [35], která je rovněž dostupná přes pip.

4.3 Konfigurace klienta (middleware) Raspberry

Jak již bylo řečeno dříve tak jako klient (middleware) vrstva bylo vybráno zařízení Raspberry Pi 3 Model B+. Jedná se o malý (ve velikosti platební karty) jednodeskový počítač běžící na architektuře ARMv8-A (64/32-bit). Procesor obsažen v Raspberry je čtyřjádrový ARM Cortex-A53 běžící na frekvenci 1.4 GHz a paměť SDRAM má velikost 1 GB. Toto jsou pouze základní parametry Raspberry, ale podrobnosti a přesné specifikace můžeme vyčíst na oficiální stránce magazínu o Raspberry [36]. Pro aplikaci v testovacím prostředí je důležité, že obsahuje USB port pro připojení Dobota a síťovou kartu pro komunikaci se serverem.

Jako vhodný obraz pro instalaci operačního systému byla vybrána oficiální distribuce **Raspbian** ve verzi 4.4.11, která vychází z Linuxové distribuce Debian. Distribuce **Raspbianu** v základu obsahuje verzi Pythonu v2.7 a Python3 ve verzi 3.4. Pro použití dostupných knihoven je však vyžadována verze minimálně 3.7. Vzhledem k tomu, že poslední oficiálně podporovaná verze Pythonu3 pro **Raspbian** je verze 3.4, musíme povýšit tuto verzi ručně. Dále je potřeba stáhnout OpenSSL ve verzi 1.1 a vyšší (ve verzi **Raspbianu** je verze 1.0 a vyšší není oficiálně podporována). OpenSSL ve verzi 1.1 a vyšší musíme stáhnout pro podporu pip, který využívá pro přístup k repositáři právě **SSL**.

4.3.1 Instalace OpenSSL 3.0.0 a Pythonu 3.8.0

Pro instalaci OpenSSL je potřeba nejprve zjistit kde na oficiálních stránkách se archív nachází a poté v Terminálu zadat příkaz, v našem případě:

```
sudo wget https://www.openssl.org/source/openssl
-3.0.0.tar.gz
```

Dále je potřeba archiv rozbalit za pomocí příkazu:

```
sudo tar xzf openssl-3.0.0.tar.gz
```

Dále je zapotřebí ve složce kde máme archiv rozbalený vygenerovat makefile:

```
sudo ./configure
```

Jako poslední krok je potřeba provést instalaci OpenSSL ve verzi 3.0.0:

```
sudo make
sudo make install
```

Po instalaci se může objevit chyba:

```
openssl: error while loading shared libraries: libssl.so.3
cannot open shared object file: No such file or directory
```

Pro odstranění chyby je potřeba provést náhradu a kontrolu verze ssl:

```
ln -s libssl.so.3 libssl.so
sudo ldconfig
openssl -V
```

Pro instalaci Pythonu 3.8.0 je potřeba opět zjistit umístění archivu na internetu a zadat příkaz v Terminálu, v našem případě:

```
sudo wget https://www.python.org/ftp/python/3.8.0/Python
-3.8.0.tgz
```

Dále je potřeba archiv rozbalit za pomocí příkazu:

```
sudo tar xzf Python-3.8.0.tgz
```

Před generováním makefile a instalací je nutné splnit prerekvizity – instalace potřebných knihoven:

```
sudo apt-get install -y build-essential tk-dev
libncurses5-dev libncursesw5-dev libreadline6-dev
libdb5.3-dev libgdbm-dev libsqlite3-dev libssl-dev
libbz2-dev libexpat1-dev liblzma-dev
zlib1g-dev libffi-dev tar wget vim
```

Dále je zapotřebí ve složce kde máme archiv rozbalený vygenerovat makefile:

```
sudo ./configure --enable-optimizations
```

Jako poslední krok je potřeba provést instalaci Pythonu ve verzi 3.8.0:

```
sudo make -j 4
sudo make altinstall
```

Správnost instalované verze můžeme ověřit:

```
python3.8 -V
```

Do Pythonu je potřeba nainstalovat balíčkoví systém pip za pomocí příkazu:

```
sudo apt-get install python3 -pip
```

V této fázi je nainstalována jak potřebná verze OpenSSL a verze Pythonu s balíčkovím systémem pip pro instalaci knihoven s možností je využívat.

4.3.2 Instalace knihoven pydobot, pymodbus, cpppo

Při instalaci knihoven je potřeba volat python3 (určení verze) -m (načtení modulu pythonu3) pip (balíčkovací systém) install název knihovny.

Instalaci knihovny **pydobot** provedeme příkazem do Terminálu:

```
sudo python3 -m pip install pydobot
```

Instalaci knihovny **pymodbus** provedeme příkazem do Terminálu:

```
sudo python3 -m pip install pymodbus[twisted]
```

Instalaci knihovny **cpppo** provedeme příkazem do Terminálu:

```
sudo python3 -m pip install cpppo
```

4.3.3 Konfigurace sítě a xRDP

Pro komunikaci se serverem je zapotřebí, aby klient i server byli na stejné síti. Pro naše potřeby byla vybrána síťová adresa ze třídy C (naše síť bude obsahovat pouze dva uzly) a to 192.168.0.30, která musí být na Raspberry nastavena jako trvalá. Toto nastavení jde provést v systémové konfiguraci v kategorii „síťové karty“. Dále je vhodné pro konfiguraci klienta nainstalovat xRDP (vzdálené ovládání), aby nebylo zapotřebí připojovat k Raspberry zobrazovací jednotku. Pro funkční připojení k vzdálené ploše přes xRDP je zapotřebí na klientovi přiřadit v uživatelskému účtu ještě heslo. V našem případě je nastaven na klientovi uživatelský účet **pi** a heslem **tom**.

Pro instalaci xRDP zadáme příkazy v Terminálu:

```
sudo apt-get update
sudo apt-get install xrdp
```

V tomto kroku je konfigurace sítě dokončena a ke klientovi se lze připojit přes vzdálenou plochu.

4.4 Konfigurace serveru

Jako server může sloužit libovolný počítač, který obsahuje Python interpreter ve verzi minimálně 3.7. V našem případě byla nainstalovaná Linuxová distribuce Ubuntu ve verzi 18.04. Dále musí být nastavena IP adresa, aby byl server ve stejné síti s klientem. Proto byla nastavena adresa 192.168.0.10. A po propojení s klientem přes Ethernet by mohlo být ověřeno spojení např. za pomocí příkazu ping na adresu 192.168.0.30. Po ověření komunikace mohla být přes základní linuxovou aplikaci Remmina ověřena funkčnost připojení ke vzdálené ploše na klienta za pomocí cílové adresy, uživatelského jména a hesla. Instalace verze Pythonu a OpenSLL nebylo zapotřebí, protože vyšší verze než potřebné minimum jsou obsaženy již v prvotní instalaci systému. Jediné co bylo potřeba doinstalovat byl balíčkoví systém pip a knihovna pymodbus a cpppo. Protože server nekomunikuje přímo s Dobotem, ale pouze s klientem tak instalace knihovny pydobot nebyla potřeba. Instalace probíhala totožně

jako na straně klienta. V poslední řadě bylo zapotřebí na server doinstalovat aplikace pro analýzu sítového provozu.

4.4.1 Instalace Wireshark

Byla vybrána aplikace Wireshark, která funguje jako analyzátor protokolů a paketový sniffer, který se nejčastěji využívá pro analýzu komunikace

Z dostupných verzí byla vybrána verze 2.6.10 pro Ubuntu. Instalaci provedeme za pomocí příkazu:

```
sudo apt-get install wireshark
```

Protože se po instalaci z důvodu omezení oprávnění nezobrazují žádné porty, je zapotřebí tento problém vyřešit příkazem:

```
sudo usermod -a -G wireshark (dolar)USER
```

A provést restartování systému pro načtení nových oprávnění pro uživatele:

```
sudo reboot
```

Po dalším přihlášení aplikace funguje korektně a je možné sledovat veškeré porty.

4.5 Implementace protokolů

Aby mohla komunikace mezi klientem a serverem fungovat, musí být na straně jak klienta, tak serveru implementovány potřebné protokoly. Jelikož v našem momentálním testovacím prostředí je obsažena jedna robotická paže Dobot s instalovaným koncovým zařízením typu vakuová přísavka, bude přes protokoly zapotřebí přenášet data, která budou určovat polohu v osách x, y, z, r a hodnotu zda-li je vakuová přísavka spuštěna nebo ne.

4.5.1 Modbus TCP

Bylo zvoleno 5 registrů, které jsou alokovány od první adresy 0x0, kdy každý registr obsahuje hodnotu daného parametru (x, y, z,..).

Strana server

Abychom mohli využívat objekty knihovny **pymodbus**, musíme je nejdříve naimportovat což provedeme za pomocí příkazu v hlavičce skriptu:

```
from pymodbus.server.asynchronous import StartTcpServer
```

Je improtován objekt z knihovny pymodbus, který nám umožní vytvořit novou instanci Modbus serveru.

```
from pymodbus.device import ModbusDeviceIdentification
```

Umožnuje nastavení identifikace serveru v rámci protokolu Modbus.

Kvůli potřebě periodického obnovování contextu je potřeba, aby aplikace byla více vláknová (je potřeba klientovi dávat periodicky jiná data). Knihovna umožňuje spuštění serveru s delegát metodou (je předána reference na funkci, která bude obnovovat stav registrů).

```
time = 2
```

Časový interval po kterém se zavolá delegát metoda, která obnovuje server context.

```
loop = LoopingCall(f=updating_writer, a=(context,))
```

Je vytvořen nový objekt, kterému jsou v konstruktoru předány parametry contextu a metody, která provádí úpravu registrů.

```
loop.start(time, now=False)
```

Je spuštěno nové vlákno, které každé 3 sekundy zavolá metodu **updating_writer**.

```
StartTcpServer(context, identity=identity, address=
("192.168.0.10", 5020))
```

Na primárním vlákně je spuštěn TCP server s daným contextem na adrese 192.168.0.10 a portu 5020.

```
def updating_writer(a):

  context = a[0]
  register = 5
  slave_id = 0x00
  address = 0x0
  next_val = next(attrs)
  print("new values: " + str(next_val))
  context[slave_id].setValues(register, address, next_val)
```

Je nastaven **context** klientovi, je definován počet zasílaných registrů a do těchto registrů je uložen následující pohyb, který je vrácen iterátorem next val.

Další funkcionality a lokalizace daných pohybů pro aplikaci přesouvání balíčků do děr dle barvy jsou zakomentovány v kódu přílohy **UpdatingServer.py**.

Strana klient

Stejně jako na straně serveru musíme objekty knihovny **pymodbus** naimportovat, což provedeme pomocí příkazu v hlavičce skriptu:

```
from pymodbus.client.sync import ModbusTcpClient as
ModbusClient
```

Umožní vytvořit instanci klienta nad protokolem Modbus.

```
from serial.tools import list_ports
```

Z knihoven serial je umožněno použití seznamu portů pro komunikaci s Dobotem.

```
import time
```

Je potřeba objekt time pro vytvoření prodlevy mezi žádostmi vůči Modbus serveru.

```
from pydobot import Dobot
```

Z knihovny **pymodbus** je importován objekt, který nám umožní vytvořit připojení k Dobotu.

```
client = ModbusClient('192.168.0.10', port=5020)
```

Je vytvořena nová instance třídy **ModbusClient** kdy jsou konstruktoru předány dva parametry - ip adresa serveru 192.168.0.10 a port serveru 5020.

```
client.connect()
```

Naváže připojení k serveru.

```
address = 0x0
```

Adresa prvního registru.

```
for i in range(100):
op = read_operation(client, address)
if op == "move_to":
  attrs = read_n_registers(client, address, 5)
  _, x, y, z, r = attrs
  device.move_to(x, y, z, r, wait=True)
elif op == "suck_it":
  attrs = read_n_registers(client, address, 2)
  device.suck(bool(attrs[1]))
```

Je vykonán cyklus o 100 pohybech. První číslo udává o jakou operaci jde. Operace jsou v kodu definovány dvě, a to „move_to“ nebo „suck_it“. Pokud jde o „move_to“ tak víme, že je potřeba načíst další příznaky a nastavit souřadnice Dobota. Pokud jde o „suck_it“ tak víme, že hodnota je pouze jedna s příznakem operace (sání nebo vypnutí sání).

```
def read_operation(client, address):
 result = client.read_holding_registers
(address, 1, unit=1)
 result_operation = {
    1: "move_to",
    2: "suck_it"
    }
 return result_operation[result.registers[0]]

def read_n_registers(client, address, n):
 result = client.read_holding_registers
(address, n, unit=1)
 result_register = []
 for i in range(n):
   result_register.append(result.registers[i])
 return result_register
```

Počet načtených registrů je závislý na operaci „move_to“ (pohyb) nebo „suck_it“ (ovládání vakuové přísavky), které jsou uloženy do asociativního pole s názvem proměnné result.

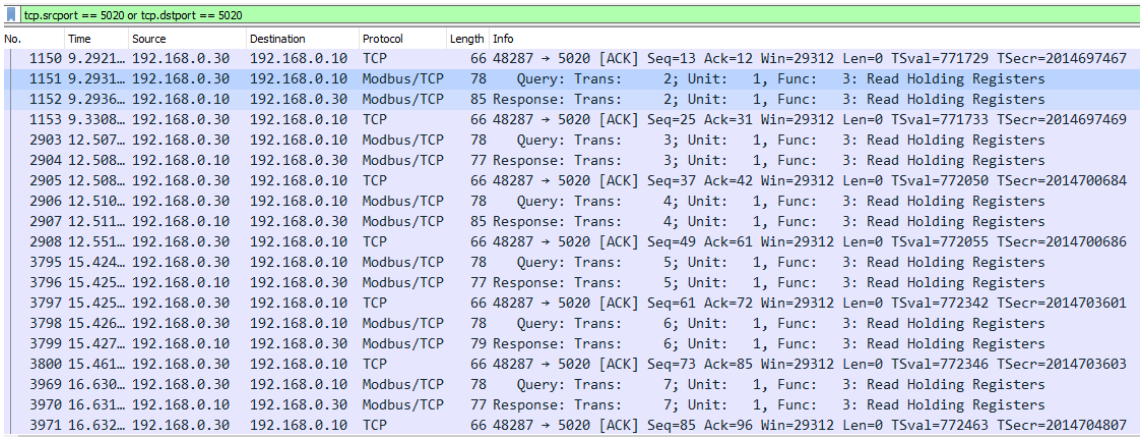
Další části kódu s komentáři jsou popsány v příloze v souboru **client.py**.

Analýza nástrojem Wireshark

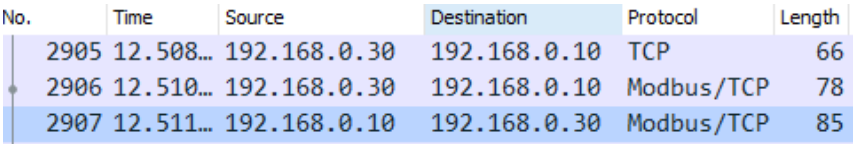
Analýza bude probíhat programem Wireshark, který byl v rámci přípravy serveru nainstalován. Zobrazení přenosu jakožto snímek obrazovky můžeme vidět na obrázku 4.8.

Na obrázku 4.9 můžeme vidět, že komunikace po protokolu **Modbus TCP** probíhá mezi klientem na adrese 192.168.0.30:5020 a serverem na adrese 192.168.0.10:5020.

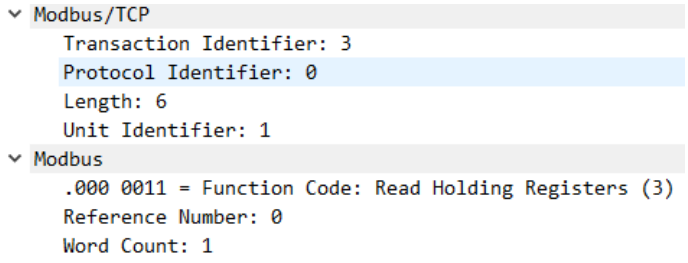
Klient si od serveru vyžádá nejprve hodnotu nultého registru viz obrázek 4.10. V závislosti na přijatých datech od serveru mohou nastat dvě situace. Buď je hodnota nultého registru 1, která definuje operaci „move_to“ viz obrázek 4.11 a nebo je hodnota nultého registru 2, která definuje operaci „suck_it“ viz obrázek 4.12.



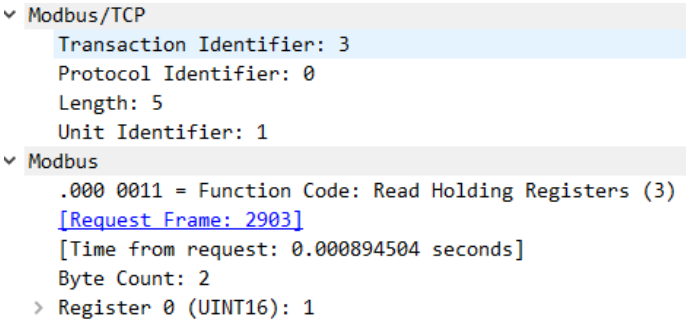
Obr. 4.8: Snímek obrazovky z aplikace Wireshark.



Obr. 4.9: Komunikace mezi klientem a serverem.

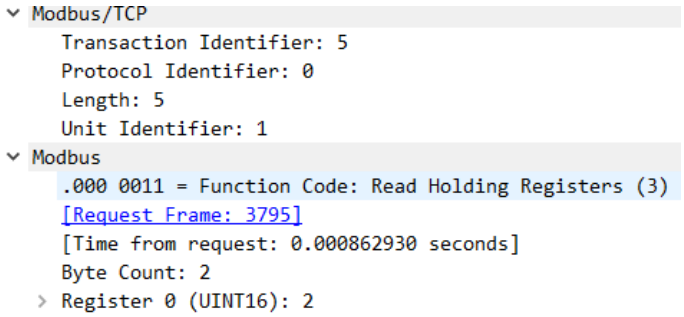


Obr. 4.10: Žádost klienta o hodnotu z registru 0.



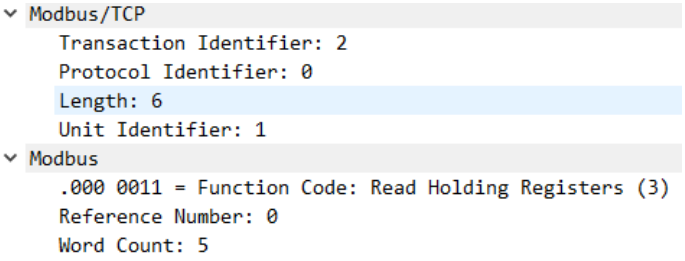
Obr. 4.11: Odpověď serveru definující příkaz „move_to“.

Pokud klient příjme od serveru registr 0 s hodnotou 1 tak si vyžádá od serveru hodnotu prvních pěti registrů viz obrázek 4.13. Registr 0 definuje příkaz „move_to“



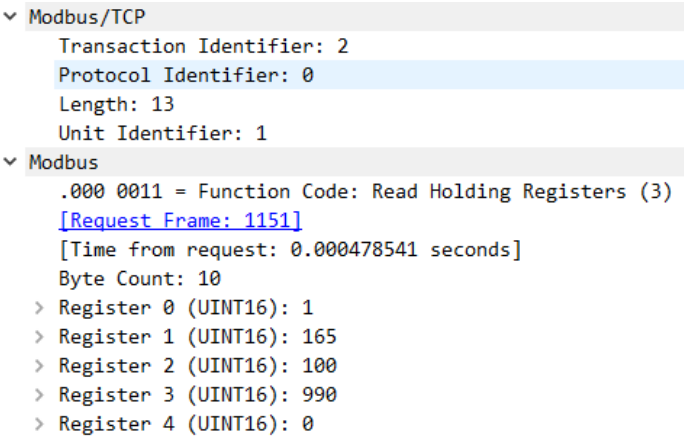
Obr. 4.12: Odpověď serveru definující příkaz „suck_it“.

a další registry definují pohyb v osách x, y, z, r.



Obr. 4.13: Žádost klienta o pět registrů.

Odpověď zaslanou klientovi od serveru v rámci operace „move_to“ lze vidět na obrázku 4.14.



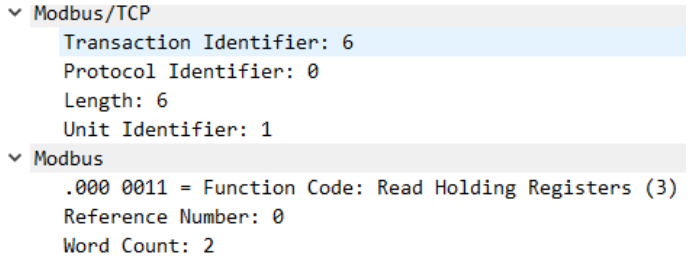
Obr. 4.14: Odpověď serveru s hodnotami registrů 0–4.

Rozbor registrů operace „move_to“:

- **Registr 0:** Nabývá hodnoty 1 a definuje operaci „move_to“ (pohyb robotické paže).

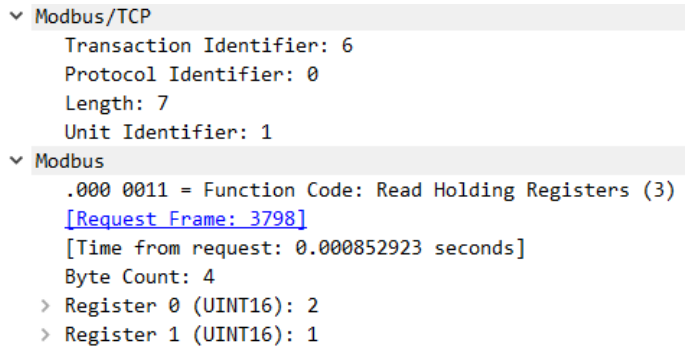
- **Registr 1:** Hodnota registru 1 definuje pohyb po ose x.
- **Registr 2:** Hodnota registru 2 definuje pohyb po ose y.
- **Registr 3:** Hodnota registru 3 definuje pohyb po ose z.
- **Registr 4:** Hodnota registru 4 definuje pohyb po ose r.

Pokud klient příjme od serveru registr 0 s hodnotou 2 tak si vyžádá od serveru hodnotu prvních 2 registrů viz obrázek 4.15. Registr 0 definuje příkaz „suck_it“ a další registr definuje pouze zapnutí nebo vypnutí vakuové přísavky.



Obr. 4.15: Žádost klienta o 2 registry.

Odpověď zaslanou klientovi od serveru v rámci operace „suck_it“ lze vidět na obrázku 4.16.



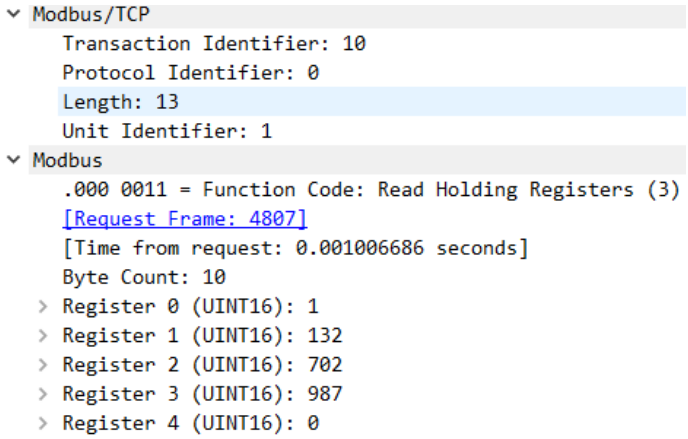
Obr. 4.16: Odpověď serveru s hodnotami registrů 0–1.

Rozbor registrů operace "suck_it":

- **Registr 0:** Nabývá hodnoty 1 a definuje operaci „suck_it“ (ovládání vakuové přísavky).
- **Registr 1:** Hodnota registru 1 může nabývat hodnoty 1 – zapni sání vakuové přísavky a nebo 0 – vypni.

Přenos dat přes protokol Modbus TCP probíhá v hexadecimální formě. Na straně serveru jsou dekadická data zakódována do hexadecimální podoby a na straně klienta

jsou opět převedena do dekadické. Příklad přenosu registru 1 s hodnotou osy y 132 je kódován do hexadecimální podoby 0084 viz obrázek 4.17.



Obr. 4.17: Dekadické číslo 132 reprezentováno jako hexadecimální.

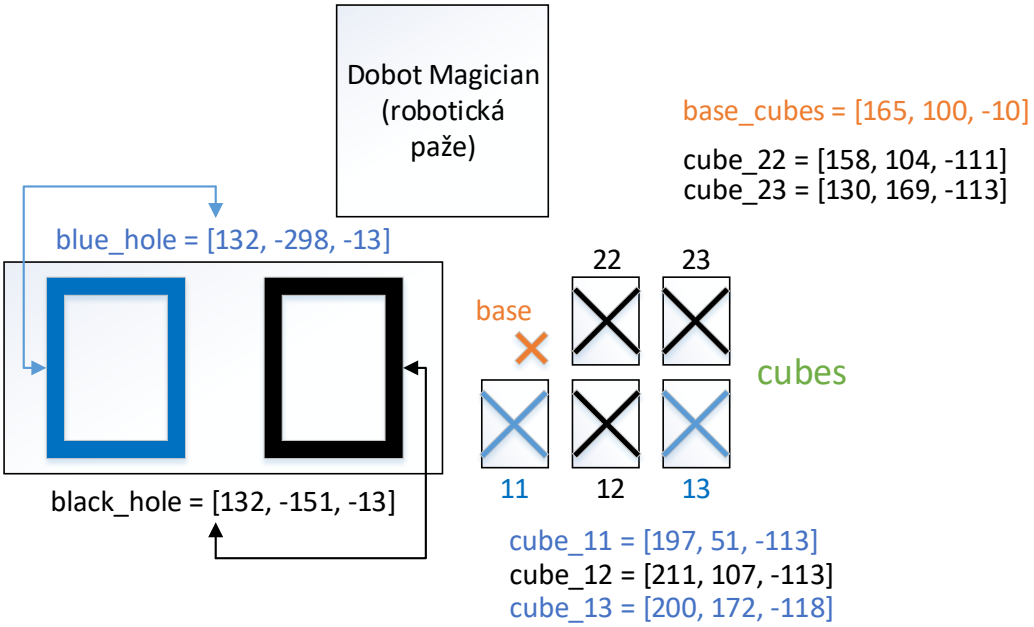
Ukázka možnosti implementace

V závislost na dostupném příslušenství byla navrhnuta ukázka implementace možného řešení. Řešení bylo přizpůsobeno aktuálním možnostem, a to tak, že nyní můžeme definovat polohu robotické paže a ovládat vakuovou přísavku. Jako ukázka byla navržena úloha pro přesun bloků do dvou děr, modré a černé v závislosti na barvě kříže na hřbetě bloku. Schéma úlohy s předdefinovanými pozicemi v osách [x, y, z] můžeme vidět na obrázku 4.18.

Pro jednoduší syntaxi programu byli předdefinovány pohyby ve formě listu:

```
base_hole = [1, 132, -151, -10, 0]
base_cubes = [1, 165, 100, -10, 0]
blue_hole = [1, 132, -298, -13, 0]
black_hole = [1, 132, -151, -13, 0]
cube_11 = [1, 197, 51, -113, 0]
cube_12 = [1, 211, 107, -113, 0]
cube_13 = [1, 200, 172, -118, 0]
cube_22 = [1, 158, 104, -111, 0]
cube_23 = [1, 130, 169, -113, 0]
suck_on = [2, 1]
suck_off = [2, 0]
```

Kdy první místo definuje operaci 1 – „move_to“ a 2 – „suck_it“. U operace pohybu jsou dále definovány polohy – [poloha v ose x, poloha v ose y, poloha v ose z, poloha v ose r] a u operace vakuové přísavky [1 – saj, 0 – vypni sání].



Obr. 4.18: Schéma rozložení pozic.

Pro definici více pohybu byl definován soubor po sobě jdoucích pohybů z listu:

```
def move_it_to_black_hole():
    return [
        suck_on, base_cubes, black_hole, suck_off
    ]
```

Přesun bloku do černé díry.

```
def move_it_to_blue_hole():
    return [
        suck_on, base_cubes, blue_hole, suck_off
    ]
```

Přesun bloku do modré díry.

```
def move_to_cube(cube):
    return [
        base_cubes, cube
```

Přesun k danému bloku.

Program je vykonán v následujícím sledu:

```
def get_next():
    arr = []
```

```
arr.extend(move_to_cube(cube_11))
arr.extend(move_it_to_blue_hole())
arr.extend(move_to_cube(cube_12))
arr.extend(move_it_to_black_hole())
arr.extend(move_to_cube(cube_13))
arr.extend(move_it_to_blue_hole())
arr.extend(move_to_cube(cube_22))
arr.extend(move_it_to_black_hole())
arr.extend(move_to_cube(cube_23))
arr.extend(move_it_to_black_hole())
```

Výsledek celého programu je demonstrován na videu v příloze.

5 Implementace průmyslové smyčky

Posledním úkolem v této práci bylo vytvoření průmyslové smyčky – v tomto případě nekonečný cyklus skládání produktu a jeho následného rozkladu. Pro tvorbu průmyslové smyčky bylo dokoupeno následující příslušenství včetně dvou dalších Dobotů Magician – Conveyor Belt Kit (pásový dopravník), který obsahuje Color sensor (senzor barev) a Sliding Rail Kit (kolejnice pro posuv jednoho Dobota). Dále byl dodán přepínač a 3 Raspberry Pi ve verzi 3B+.

5.1 Návrh průmyslové smyčky

Průmyslová smyčka byla navržena tak, aby mohlo být otestováno a využito veškeré dokoupené příslušenství a zároveň, aby byl splněn předpoklad nekonečného opakování. Veškerá manipulace s předměty je realizována za pomocí přísavky.

Obecný popis chování průmyslové smyčky – rozděleno na skladbu a rozklad (tím je vytvořen nekonečný cyklus):

Skladba:

- **Krok 1:** Dobot s ID1 nabere krabičku a vloží ji na pás. Dále odjede po kolejnici téměř nakonec.
- **Krok 2:** Dobot s ID2 nabere výplň a vloží ji do krabičky. Dále Dobot s ID3 posune pás s krabičkou a výplní k pozici senzoru barev.
- **Krok 3:** Dobot s ID2 nabere jednu ze 4 kostek (náhodně) a za pomocí senzoru barev otestuje její barvu – pokud je barva jiná než zelená tak vrátí kostku na původní místo, pokud je barva kostky zelená vloží ji do krabičky. Dále Dobot s ID3 posune pás s krabičkou na konec pásu.
- **Krok 4:** Dobot s ID1 vloží žlutou kostku do krabičky a Dobot s ID3 nasadí kryt na krabičku – tím je ukončen cyklus složení.

Rozklad:

- **Krok 1:** Dobot s ID3 sejme kryt z krabičky a vrátí jej na původní místo a Dobot s ID1 vezme z krabičky žlutou kostku a vrátí ji na původní místo. Dále se Dobot s ID1 vrátí po kolejnici na počáteční pozici a Dobot s ID3 posune pás k pozici senzoru barev.
- **Krok 2:** Dobot s ID2 vrátí zelenou kostku na pozici ze které ji vzal a Dobot ID3 krabičku na pásu na začátek.
- **Krok 3:** Dobot s ID2 vrátí výplň na původní pozici a Dobot s ID1 vrátí krabičku na původní pozici – tím je ukončen cyklus rozložení.

Návrh průmyslové smyčky můžeme vidět v příloze A na obrázku A.1.

5.2 Realizace průmyslové smyčky

Pro skutečnou realizaci průmyslové smyčky je zapotřebí se nejprve seznámit s novým příslušenstvím a nakonfigurovat nové, školní Raspberry Pi. Jako meziprvek pro propojení komunikace může být použit jakýkoliv přepínač s minimálně 4 porty.

5.2.1 Příslušenství k Dobot Magician

Dokoupené příslušenství k Dobot Magician zahrnuje Sliding Rail Kit a Conveyor Belt Kit jehož součástí je i Color sensor.

Sliding Rail Kit

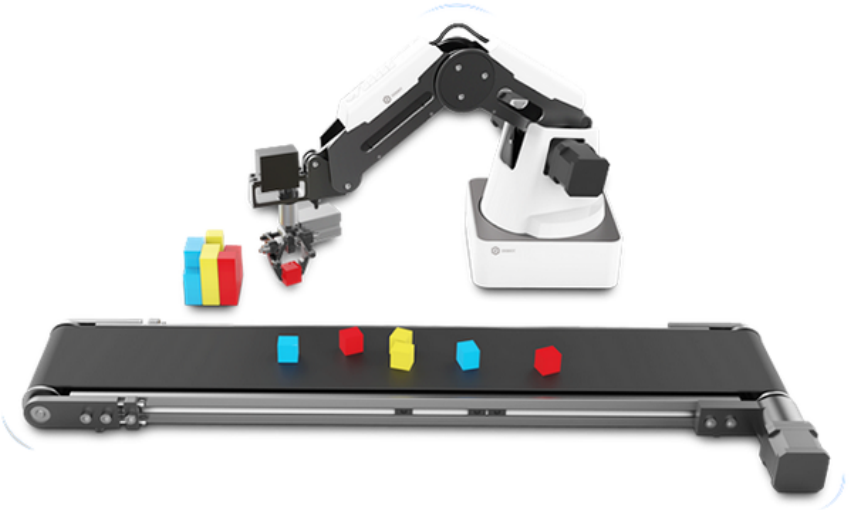
Sliding Rail Kit umožnuje robotické paži Dobot Magician pohyb po railu (v ose L) v rozsahu 0–1000 mm. Tím je zajištěna mobilita Dobota a větší rozsah pohybu. Vizualizaci Dobota umístěného na kolejnici můžeme vidět na obrázku 5.1 a náhled do dokumentace daného produktu na adrese [38].

Obr. 5.1: Ukázka Sliding Rail Kitu [37].

Conveyor Belt Kit

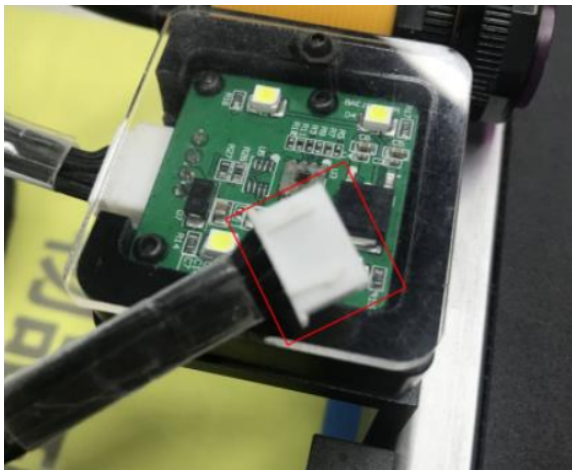
Conveyor Belt Kit umožňuje pohyb předmětů po páse v obou směrech v rozsahu 700 mm při definované rychlosti a zrychlení. Tudíž je možné na pás umisťovat přemety a dle libosti je posunovat ke zpracování. Ukázku Conveyor Beltu můžeme vidět na obrázku 5.2 a náhled do dokumentace daného produktu na adrese [40].

Součástí Conveyor Belt Kitu je i Color sensor, který je vybaven přísvitem 4 LED diodami a v závislosti na jeho firmwaru umí vracet pouze barvy červenou, zelenou



Obr. 5.2: Ukázka Conveyor Beltu [39].

a modrou ve formě [1/0, 1/0, 1/0]. Ukázku Color sensoru můžeme vidět na obrázku 5.3 a dokumentace je součástí dokumentace celého kitu.



Obr. 5.3: Ukázka Color sensoru [39].

5.2.2 Konfigurace klientů a serveru

Univerzitou byly dodány nové Raspberry Pi ve verzi 3B+, a ačkoli se jednalo o stejnou verzi, jako byla využita v testovacím prostředí, tak nebylo možno použít naklonovat původní Raspberry Pi – nejednalo se o stejnou revizi.

Konfigurace klientů

Bylo zapotřebí použít čistou instalaci operačního systému Raspberry Pi OS ve verzi 4.19.97 stažitelného z odkazu [41]. Poté stačí splnit pouze prerekvizity z kapitoly 4.3 a část klienta je nakonfigurována. Následně je možnost klonovat daného klienta na další Raspberry Pi – k tomuto je možno využít programu Win32DiskImager (pro Windows) stažitelného z adresy [42]. V rámci jiných operačních systémů, jako je Linux nebo MacOS, lze použít příkaz "dd". Klienti byli označení vizuálně jejich IP adresou a názvem rPI 1–3, aby nemohlo dojít k jejich záměně v laboratoři.

Protože nyní máme oproti testovacímu prostředí rPI ve 3 kusech je potřeba nastavit jejich IP adresy (včetně účtu uživatel:heslo), a přiřadit je k příslušné robotické paži Dobot:

- **rPI1** – Ovládá Dobota s ID1, IP adresa: 192.168.0.30, uživatel: pi, heslo: tom.
- **rPI2** – Ovládá Dobota s ID2, IP adresa: 192.168.0.31, uživatel: pi, heslo: tom.
- **rPI3** – Ovládá Dobota s ID2, IP adresa: 192.168.0.32, uživatel: pi, heslo: tom.

Z původní konfigurace rPI je možno se k rPI 1-3 připojovat za pomocí xRDP protokolu. Nově byla přidána pro lepší manipulaci s klienty možnost se připojovat přes SSH protokol za pomocí příkazů:

```
ssh pi@192.168.0.30
ssh pi@192.168.0.31
ssh pi@192.168.0.32
```

Konfigurace serveru

Oproti testovacímu prostředí byl jako server využit notebook MacBook Pro 15 (2015) s aktuální verzí operačního systému MacOS Catalina již bez virtuálního prostředí Linuxu. Bylo zapotřebí splnit pouze prerekvizity z instalace serveru uvedené v kapitole 4.4. Avšak je možné využít libovolný notebook s OS Linux, popř. další Raspberry Pi jako server.

- **Server – komunikuje s rPI 1–3** – IP adresa: 192.168.0.10

5.2.3 Úprava knihovny dobot.py

Pro ovládání Dobota byla využita knihovna pydobot [33]. Knihovna měla naimplementováno ovládání pohybů ramene a vakuové přísavky. Pro ovládání periferií Rail, Color sensor a Convoyer belt bylo potřeba knihovnu upravit. Jelikož knihovna má veřejně dostupné zdrojové kódy tak jsme udělali úpravy přímo v ní.

Nutné úpravy byly provedeny dle dokumentace Development protokolu – Dobot Communication Protocol v1.1.5 uvedeném na stránkách [43] viz příloha B.

Rail

Dobot s ID1 je vybaven periferií rail, která posouvá Dobotem po kolejnici. Pohyb je prováděn metodou „move_to_withL(x, y, z, r, l)“, která je vytvořena dle dokumentace [43], a má identifikátor operace ID=86 [43] na straně 39 v kapitole 1.10.7 – SetPTPWithLCmd. Datagram řídícího paketu SetPTPWithLCmd můžeme vidět na obrázku 5.4.

Header	Len	Payload				Checksum
		ID	Ctrl		Params	
			rw	isQueued		
0xAA 0xAA	2+21	86	1	1 or 0	PTPWithLCmd（See Program 19）	Payload checksum

Obr. 5.4: Datagram SetPTPWithLCmd [43].

Parametry „Params“ byly naplněny souřadnicemi x, y, z, r a přibyl k nim parametr l, který udává pohyb po kolejnici.

Před použitím metody „movie_to_withL“ je zapotřebí na Dobotovi aktivovat připojenou periferii rail. Dle dokumentace se kolejnice aktivuje identifikátorem operace ID=3na straně 13 v kapitole 1.3.4 – Set/Get DeviceWithL. Datagram řídícího paketu Set DeviceWithL můžeme vidět na obrázku 5.5 [43]. V námi upravené knihovně se jedná o metodu „enable_rail()“.

Header	Len	Payload				Checksum
		ID	Ctrl		Params	
			rw	isQueued		
0xAA 0xAA	2+2	3	1	0	WithL withL (See Program 1)	Payload checksum

Obr. 5.5: Datagram Set DeviceWithL [43].

Parametr „Params“ naplníme polem o jednom prvku, který nabývá hodnoty True/False, jestliže je k Dobotovi připojena periferie rail.

Convoyer belt

Dobot s ID3 je vybaven periferií Convoyer belt, která umožňuje pohyb předmětů po pásovém dopravníku. Pohyb pásového dopravníku je prováděn za pomocí metody „move_conveyor()“ s parametry distance, direction a speed. Metoda je vytvořena zapouzdřením tří volání. Jedná se o volání „start_stepper(speed)“ a „stop_stepper()“, které zapínají a vypínají pohyb pásového dopravníku, a metoda „set_wait()“, která zajišťuje posuv o danou vzdálenost výpočtem času běhu pásového dopravníku.

Směr pásového dopravníku „direction“ nabývá hodnot 0 nebo 1 – je li hodnota 1, tak se dopravník pohybuje v opačném směru. Ve své podstatě jde o negování parametru „speed“. Metoda „set_wait()“ na základě parametru speed a znalosti rychlosti otáčení pásového dopravníku způsobí čekání, které udá vzdálenost posuvu.

Metody „start_stepper()“ a „stop_stepper()“ pracují se stejnou řídící metodou. Jedná se o metodu s ID=135 [43] na straně 55 v kapitole 1.15.6 – Set EMotor. Datagram řídícího paketu Set EMotor můžeme vidět na obrázku 5.6 [43].

Header	Len	Payload				Checksum
		ID	Ctrl		Params	
			rw	isQueued		
0xAA 0xAA	2+2	135	1	1 or 0	EMotor（See Program 36）	Payload checksum

Obr. 5.6: Datagram Set EMotor [43].

Color sensor

Dobot s ID2 je vybaven periferií Color sensor, která umožňuje čtení barvy předmětu. Funkčnost Color sensoru je zajištěna metodami „enable_color_sensor()“, „disable_color_sensor()“ a „read_color()“. Všechny 3 metody využívají operace s ID=137 uvedené na straně 56 v kapitole 1.15.7 – Set/Get ColorSensor. Datagram řídícího paketu Set/Get ColorSensor můžeme vidět na obrázku 5.7 [43].

Header	Len	Payload				Checksum
		ID	Ctrl		Params	
			rw	isQueued		
0xAA 0xAA	2+3	137	1	1 or 0	Device ColorSense （See Program 37）	Payload checksum

Obr. 5.7: Datagram Set/Get ColorSensor [43].

Pro metody „enable_color_sensor()“ a „disable_color_sensor()“ se naplní „Params“ parametry „state“, „port“ a „version“, přičemž „state“ reprezentuje stav senzoru. Pro získání barvy v metodě „read_color()“ se „Params“ naplní pouze parametrem „port“. Po vykonání řídícího příkazu je vrácena barva ve formátu asociativního pole, které obsahuje hodnoty „R“, „G“ a „B“.

5.2.4 Implementace průmyslových protokolů

Pro praktickou implementaci průmyslových protokolů byly vybrány následující protokoly a to **Modbus TCP** a **EtherNET/IP**. Komunikace probíhá ve formě přenosu mezi klienty a serverem.

Struktura aplikace je následující:

- **Server:** Pro oba dva protokoly je použita stejná verze. Rozdíl je jen v provedené konfiguraci.
- **Klient:** Pro každý z protokolů byla implementace provedena zvlášť.

Serverová část

Po spuštění serverové aplikace **server.py**, jsou vytvořeny instance objektů jednotlivých protokolů, které jsou jako parametry předány do hlavní smyčky aplikace. Poté je vytvořena fronta operací, které budou přenášeny na jednotlivé klienty.

Operace jsou definovány v enumerátoru a mohou nabývat hodnot:

- **move:** Pohyb robotické paže v osách x, y, z, r, l.
- **suck:** Zapnutí nebo vypnutí vakuové přísavky.
- **move_conveyor:** Manipulace s pásem.
- **read_color:** Čtení barvy.

Po přenesení operace čeká smyčka na potvrzení klientem, poté je přenesena operace další. Takto se vykonají všechny definované operace a poté se fronta naplní znovu.

V definovaných operacích může být i podmínková funkce, která mění přenášené pozice v závislosti na vstupních parametrech. Tak je definován výběr kostky správné barvy v průmyslové smyčce.

Povely pro klienta jsou přenášeny po vybraném protokolu na základě prvotní konfigurace aplikace.

Pro definici jednotlivých operací slouží třída **Arm**, která definuje všechny výše uvedené operace. Z této třídy vychází třídy pro jednotlivé instance, které mají oproti

základní třídě definované i pozice a funkce, které kontextově souvisí s daným objektem.

Pro přenos informací po protokolu **Modbus TCP** slouží třída **ModbusServer**, pro přenos po protokolu **EtherNET/IP** slouží třída **Enipcontext**. Ty obsahují metodu „update_context()“, které zajišťuje úpravu registrů na hodnoty definované na vstupu metody.

Metody přijímají parametry:

- **slave_id:** Definuje adresu klienta, pro kterého je hodnota určena.
- **values:** Asociativní pole hodnot, které budou zapsány do registrů.

Ostatní parametry funkce jsou základně nastaveny na hodnotu enumerátoru, která definuje adresu registru, do kterého budou hodnoty zapsány.

Další metodou je „is_function_finished()“, vyčítající hodnotu registru, která definuje, že klient již námi poslanou operaci zpracoval. Metoda přijímá jediný parametr „slave_id“, definující adresu klienta, kterému operace příslušela.

Poslední důležitou metodou je metoda „get_color“, přijímající parametr s názvem „slave_id“, který definuje adresu klienta a jejíž návratovou hodnotou je asociativní pole, která obsahuje informace o barvě ve formátu R, G a B.

Obě výše uvedené třídy využívají metody „encode“, která byla implementována z důvodu schopnosti protokolů přenášet pouze kladné hodnoty (datový typ jednotlivých registrů je UINT16).

Jednotlivé operace průmyslové smyčky byly uchovány ve frontě. K tomu jsme si zadefinovali třídu „CommandQueue“, kde tato třída funguje jako fronta FIFO.

Třída má metody:

- **add(cmds):** Metoda očekává na vstupu pole instrukcí, které vloží do fronty.
- **remove(index):** Metoda odstraní z fronty instrukci na pozici index a vrátí její hodnotu.
- **add_first(cmd):** Metoda vloží na začátek fronty instrukci „cmd“.
- **get_next(cmd):** Metoda vrátí instrukci z vrcholu fronty.

V těle nekonečné procedury „update_context“, vykonávající průmyslovou smyčku dochází ke čtení z fronty instrukcí. Je li fronta prázdná, zavolá se funkce „fill_queue“, která frontu naplní. V proceduře „update_context“ dochází k rozhodnutí, která paže a jakým protokolem se instrukce vykoná a přenese. Zdrojový script v jazyce python serveru je uveden v příloze C.

Klientská část

Narozdíl od serverové části musela být klientská část rozdělena v závislosti na použitém protokolu. Klient s protokolem **Modbus TCP** byl popsán v kapitole 5.2.4 a klient s protokolem **EtherNET/IP** v kapitole 5.2.4. Příloha D obsahuje zdrojové kódy scriptu Modbus TCP klienta a příloha E obsahuje zdrojové kódy scriptu EtherNET/IP klienta.

Modbus TCP – Klient

Po spuštění klienta dojde k načtení konfigurace klienta a poté proběhne přípojení po protokolu **Modbus TCP** se serverem. Dále klient zinicializuje spojení s robotickou paží Dobot. Je-li v konfiguraci řečeno, že daný Dobot obsahuje periferii rail, tak je potřeba provést její aktivaci voláním metody enable_rail. Poté se klient dostane do nekonečné smyčky, kdy čeká na instrukce od serveru.

Klient vždy čte instrukce ze serveru, jsou-li instrukce adresovány Dobotovi provede operace a odešle serveru informaci, že je klient opět v nečinném stavu. Informace, že je klient v nečinném stavu odešle klient i v okamžiku, kdy zjistí, že daná instrukce nenáleží jemu.

Čtení instrukcí ze serveru probíhá tak, že se přečte obsah prvního registru, kterým klient zjistí o jakou operaci se jedná, a kolik má přečíst dalších parametrů z registrů. Operace klienta jsou v tabulce 5.1, ve které můžeme vidět i počet vstupních parametrů.

Tab. 5.1: Operace klienta.

ID operace	**Název operace**	**Počet parametrů na vstupu**
1	MOVE	6
2	SUCK	2
3	MOVE_CONVOYER	2
4	READ_COLOR	2

Po přečtení instrukce klient rozhodne, zda-li se jedná o instrukci novou nebo již vykonanou. Jde-li o instrukci novou, klient ji zpracuje a pošle jako instrukci Dobotovi. Prvním krokem zpracování je operace „decode“, která je funkcí inverzní k operaci „encode“. Jak klient instrukce zpracuje a odešle je na Dobota můžeme vidět ve zdrojovém scriptu **client.py** 5.2.4.

EtherNET/IP – Klient

Podobně jako předchozí klient, klient pracující s protokolem **EtherNET/IP** načte z konfigurace adresu server, na kterou se připojí. Následně je inicializováno spojení s Dobotem, a dle konfigurace je připraven aktivovat periferii rail.

Narozdíl od klienta pracujícím s protokolem **Modbus TCP** se zde načte všech šest registrů, a to proto, že se jedná o maximální možný počet potřebných parametrů. Stále platí, že první z registrů nese informaci o dané operaci. Platí i nadále tabulka 5.1 s pozměněnou interpretací posledního sloupce „Počet parametrů na vstupu“, který teď udává počet registrů nesoucí užitečnou informaci. Zbývající registry mají náhodnou hodnotu a jsou ignorovány.

Základní metoda nepojení na server „client.connector“, knihovny **cpppo** [35] protokolu **EtherNET/IP** má definovaný přenos pouze datového typu „U_INT4“. To pro naše potřeby bylo značně omezující z důvodu potřeby přenášet i záporné hodnoty, nebo zakódovat hodnoty metodami „endcode/decode“ stejně, jako v případě protokolu **Modbus TCP**. Proto bylo využito pro získávání hodnot metody „proxy_simple“, kde můžeme definovat i návratový datový typ.

Zbývající interpretace a odesílání instrukcí na Dobota je identické s klientem na protokolu **Modbus TCP**. Celý script klienta je dostupny v souboru **client_ethip.py**, který můžeme vidět v příloze E.

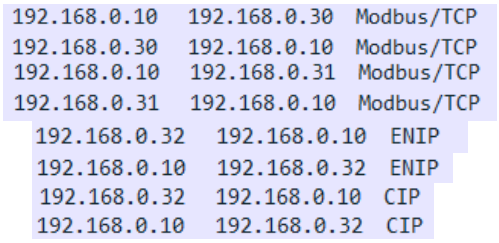
5.2.5 Analýza Wiresharkem

Analýza bude stejně, jako v případě testovacího prostředí probíhat za pomocí programu Wireshark. Zobrazení komunikace můžeme vidět jako výpis ze záznamu na obrázku 5.8. Jak je z obrázku patrné, tak komunikace obsahuje využití protokolů **Modbus TCP** a **EtherNET/IP** – (CIP + ENIP).

No.	Time	Source	Destination	Protocol	Length	Info
391	11.057553	192.168.0.30	192.168.0.10	Modbus/TCP	78	Query: Trans: 58; Unit: 1, Func: 3: Read Holding Registers
392	11.057616	192.168.0.10	192.168.0.30	TCP	66	5020 → 48101 [ACK] Seq=609 Ack=754 Win=131008 Len=0 TSval=1440063913 TSecr=1788221855
393	11.057966	192.168.0.10	192.168.0.30	Modbus/TCP	77	Response: Trans: 58; Unit: 1, Func: 3: Read Holding Registers
394	11.058369	192.168.0.30	192.168.0.10	TCP	66	48101 → 5020 [ACK] Seq=754 Ack=620 Win=64256 Len=0 TSval=1788221856 TSecr=1440063913
395	11.058848	192.168.0.30	192.168.0.10	Modbus/TCP	78	Query: Trans: 59; Unit: 1, Func: 3: Read Holding Registers
396	11.058891	192.168.0.10	192.168.0.30	TCP	66	5020 → 48101 [ACK] Seq=620 Ack=766 Win=131008 Len=0 TSval=1440063914 TSecr=1788221856
397	11.059520	192.168.0.10	192.168.0.30	Modbus/TCP	75	Response: Trans: 59; Unit: 1, Func: 3: Read Holding Registers. Exception returned
398	11.060423	192.168.0.30	192.168.0.10	Modbus/TCP	81	Query: Trans: 60; Unit: 1, Func: 16: Write Multiple Registers
399	11.060500	192.168.0.10	192.168.0.30	TCP	66	5020 → 48101 [ACK] Seq=629 Ack=781 Win=131008 Len=0 TSval=1440063915 TSecr=1788221858
400	11.060865	192.168.0.10	192.168.0.30	Modbus/TCP	78	Response: Trans: 60; Unit: 1, Func: 16: Write Multiple Registers
401	11.071848	Raspberr_01:…	Broadcast	ARP	60	Who has 192.168.0.1? Tell 192.168.0.31
402	11.085259	192.168.0.32	192.168.0.10	ENIP	94	Register Session (Req), Session: 0x00000000
403	11.085345	192.168.0.10	192.168.0.32	TCP	66	44818 → 46392 [ACK] Seq=1 Ack=29 Win=131712 Len=0 TSval=1440063939 TSecr=155442433
404	11.088501	192.168.0.10	192.168.0.32	ENIP	94	Register Session (Rsp), Session: 0x808556F5
405	11.088876	192.168.0.32	192.168.0.10	TCP	66	46392 → 44818 [ACK] Seq=29 Ack=29 Win=64256 Len=0 TSval=155442437 TSecr=1440063942
406	11.098189	192.168.0.32	192.168.0.10	ENIP	90	List Identity (Req)
407	11.098340	192.168.0.10	192.168.0.32	TCP	66	44818 → 46392 [ACK] Seq=29 Ack=53 Win=131712 Len=0 TSval=1440063951 TSecr=155442446
408	11.104655	192.168.0.10	192.168.0.32	ENIP	150	List Identity (Rsp), 1756-L61/B LOGIX5561
409	11.106058	192.168.0.30	192.168.0.10	TCP	66	48101 → 5020 [ACK] Seq=781 Ack=641 Win=64256 Len=0 TSval=1788221903 TSecr=1440063915
410	11.142289	192.168.0.32	192.168.0.10	CIP	114	Class (0x93) - Get Attribute Single

Obr. 5.8: Snímek obrazovky z aplikace Wireshark.

Server má IP adresu 192.168.0.10 a komunikuje s klienty na adresách 192.168.0.30, 192.168.0.31 a 192.168.0.32. Komunikace za pomocí protokolu **Modbus TCP** je vedena s klienty s adresami 192.168.0.30 a 192.168.0.31. Pouze klient s IP adresou 192.168.0.32 komunikuje po protokolu **EtherNET/IP**. Záznam komunikace mezi klienty a serverem můžeme vidět na obrázku 5.9.

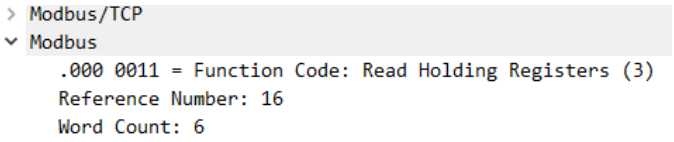


Obr. 5.9: Komunikace mezi klienty a serverem.

Komunikace za pomocí protokolu Modbus TCP

Oproti testovacímu prostředí jsou navíc přenášeny operace pro posuv kolejnice, vyčítání hodnoty z senzoru barev a potvrzení operace.

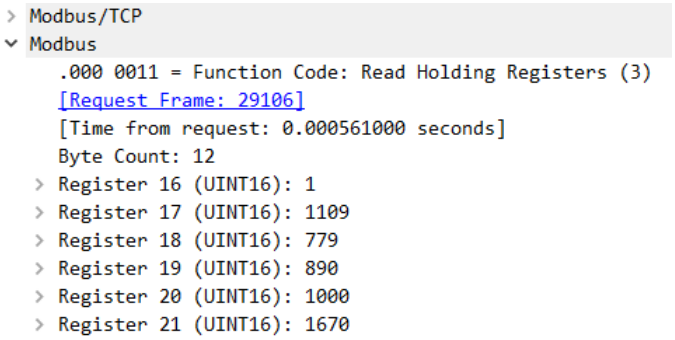
Na obrázku 5.10 můžeme vidět žádost klienta o šest registrů a jako odpověď od serveru dostane 6 po sobě jdoucích registrů, které reprezentují v nultém registru operaci „move“, a další obdržené registry pohyb v osách [x, y, z, r a l] (pohyb + posuv kolejnice). Odpověď od serveru můžeme vidět na obrázku 5.11.



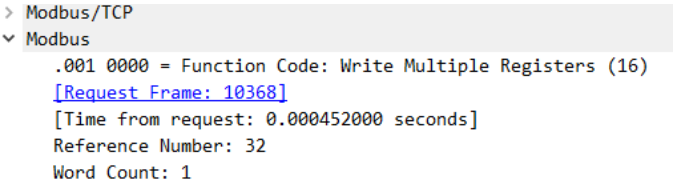
Obr. 5.10: Žádost od klienta o šest registrů.

Z obrázku můžeme vyčíst, že hodnota registru reprezentující souřadnici l, nabývá hodnoty 10670. Což po dekódování udává pohyb kolejnice o 670 mm.

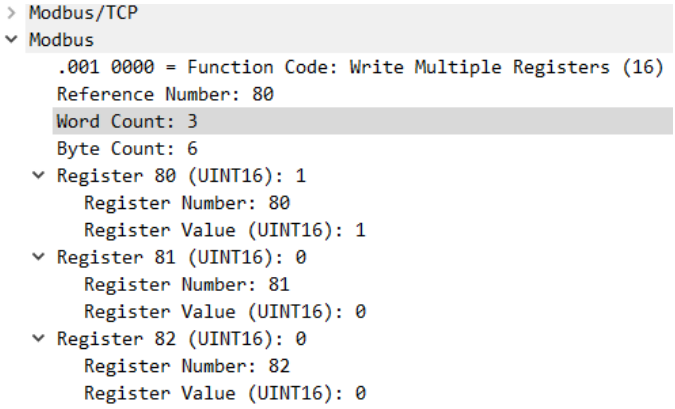
Žádost klienta o čtení hodnoty z senzoru barev můžeme vidět na obrázku 5.12. Klient od serveru dostane jeden registr a vrací mu hodnotu 3 registrů – tuto situaci můžeme vidět na obrázku 5.13. Pořadí registrů je vráceno jako „R“, „G“ a B každý registr nabývá hodnoty 0/1. V našem případě se jednalo o hodnoty [1, 0, 0] – tudíž byla vrácena červená barva.



Obr. 5.11: Odpověd serveru s hodnotou šesti registrů.



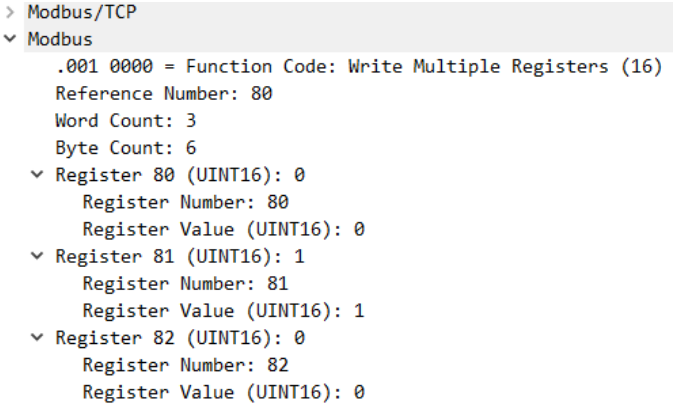
Obr. 5.12: Žádost kliena o jeden registr.



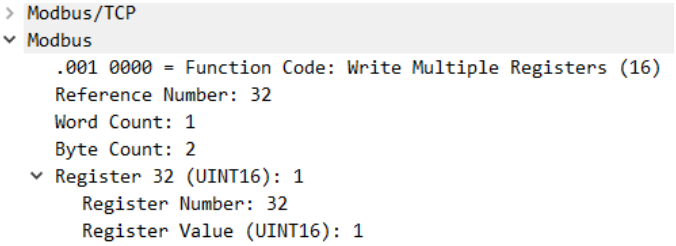
Obr. 5.13: Vrácení červené barvy.

Oproti tomu na obrázku 5.14 můžeme vidět z pořadí přijatých registrů ve formátu [0, 1, 0], že byla vrácena barva zelená.

Po každé operaci vykonané klientem, klient zapíše do registru 0x20 hodnotu 1, která dává serveru informaci, že může začít přenášet další operaci. Přenos paketu můžeme vidět na obrázku 5.15.



Obr. 5.14: Vrácení zelené barvy.



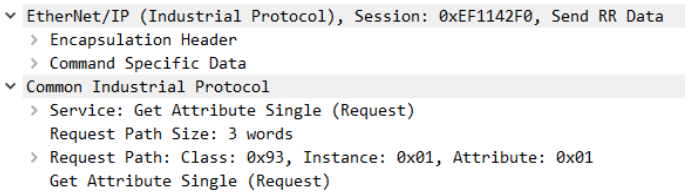
Obr. 5.15: Potvrzení vykonané operace.

Komunikace za pomocí protokolu EtherNET/IP

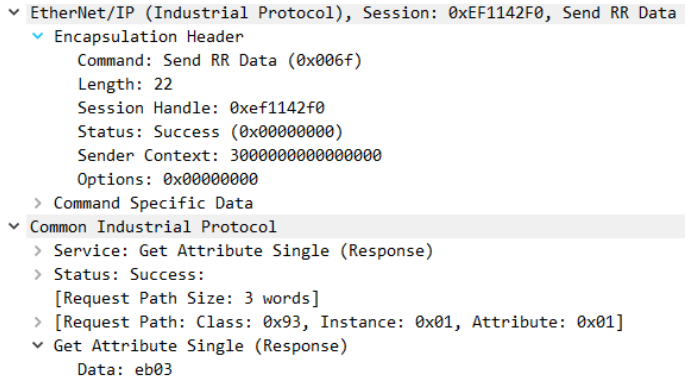
Po protokolu EthernNET/IP probíhá ovládání pásového dopravníku, pohyb paže, ovládání vakuové přísavky a potvrzování operace.

Žádost klienta o operaci můžeme vidět na obrázku 5.16. Přenesení operace ze serveru s výsledkem **3** nám udává, že se jedná o operaci pohybu pásu. Tuto situaci můžeme vidět na obrázku 5.17. Pouze je potřeba si dávat pozor, že vrácená hodnota je uvedena v hexadecimálním tvaru. Hodnota je **eb03**, tu je navíc potřeba číst jako **03eb** což v decimálním zápisu značí **1003** a po dekódování jde o operaci s číslem **3** – „move_convoyer“.

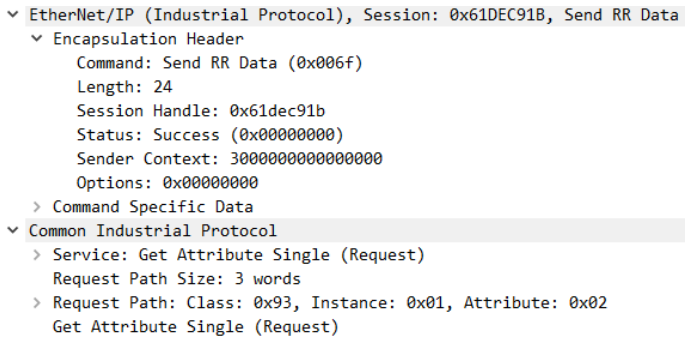
Dále si klient vyžádá od serveru hodnotu o kolik centimetrů se má s pásovým dopravníkem posunout. Tuto žádost můžeme vidět na obrázku 5.18. Server mu následně poskytne odpověď s informací hodnoty posuvu viz obrázek 5.19. V přijatém paketu dostaneme hodnotu **1004**, která se opět čte jako **0410**, což je decimálně **1040**. Po provedení dekódování dostaneme hodnotu **40**, která reprezentuje posuv o 40 cm.



Obr. 5.16: Žádost klienta o operaci.

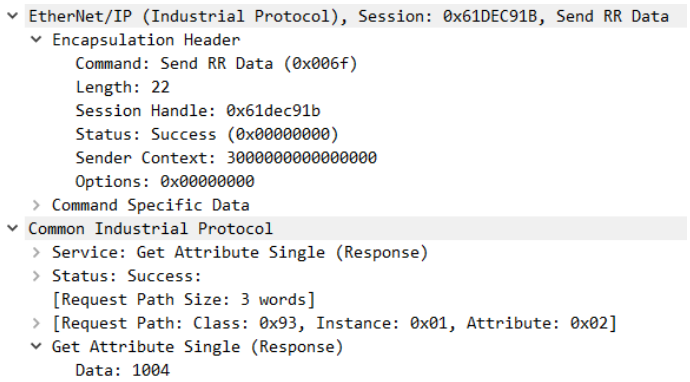


Obr. 5.17: Odpověď serveru – 3 – „move_conveyor“.

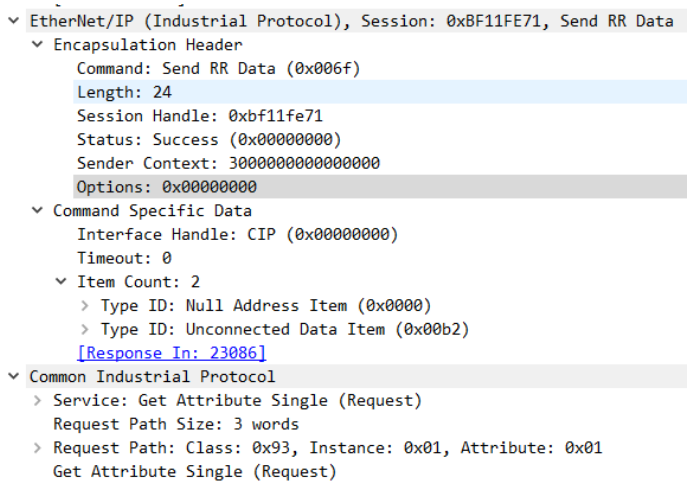


Obr. 5.18: Žádost klienta o hodnotu posuvu.

Další operací je ovládání vakuové přísavky. Klient si vyžádá hodnotu prvního registru **0x01** viz obrázek 5.20. Jako odpověď od serveru dostane hodnotu prvního registru s hexadecimální hodnotou **ea03**. Tuto hodnotu je opět potřeba číst jako **03ea**. Po převodu do dekadické soustavy dostáváme hodnotu **1002**, která po dekódování značí hodnotu číslo **2**. Tímto číslem je reprezentována operace „suck“. Odeslaný paket od serveru na klienta můžeme vidět na obrázku 5.21.



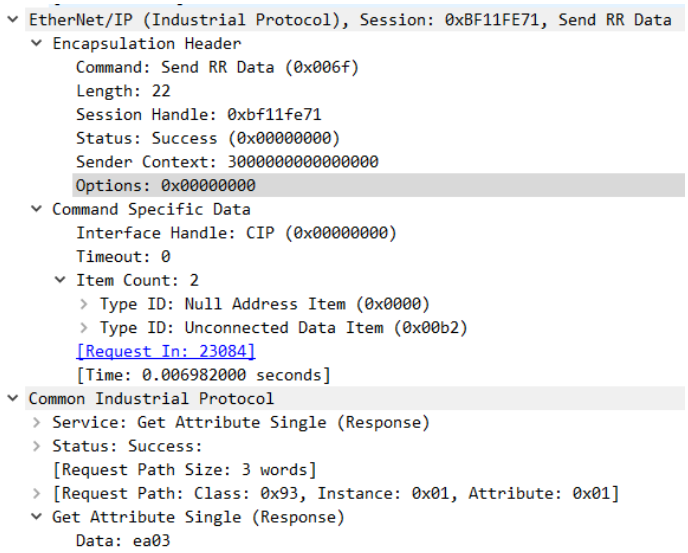
Obr. 5.19: Odpověď serveru s hodnotou posuvu – 40 cm.



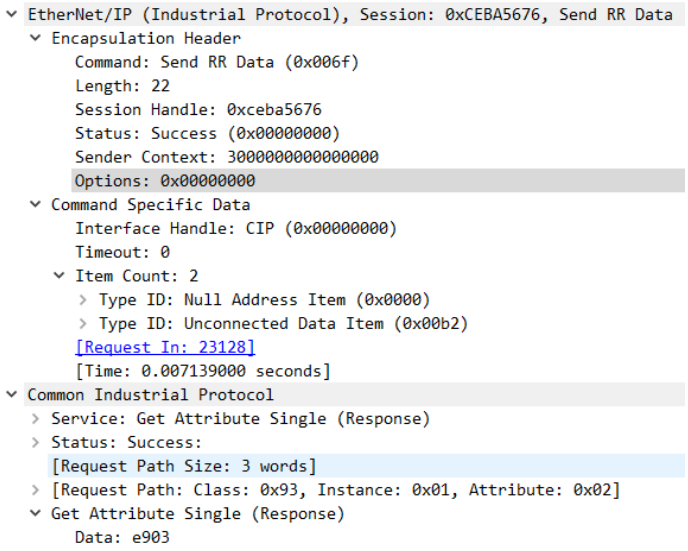
Obr. 5.20: Žádost klienta o hodnotu registru 0x01.

Následně si klient vyžádá od serveru hodnotu registru **0x02**, která nám udává zda se má vakuová přísavka zapnout či vypnout. Pro demonstraci bylo vybráno zapnutí vakuové přísavky, které můžeme vidět na obrázku 5.22. Klient obdrží od serveru v registru **0x02** hodnotu **e903**, která se opět čte jako **03e9**. Po převodu do dekadické podoby se jedná o číslo **1001**, a po dekódování získáme hodnotu **1**. Hodnota **1** reprezentuje zapnutí sání. Pokud by byla odpověď od serveru po převodu **0**, tak by se jednalo o vypnutí sání.

Ovládání pohybu je definováno operaci „move“, kdy si klient vyžádá od serveru hodnotu registru **0x01** jak můžeme vidět na obrázku 5.23. Jako odpověď od serveru dostane hexadecimální hodnotu **e903**, která je čtena jako **03e9** a po převodu do dekadické soustavy značí číslo **1001**. Po dekódování dostáváme číslo operace **1**,



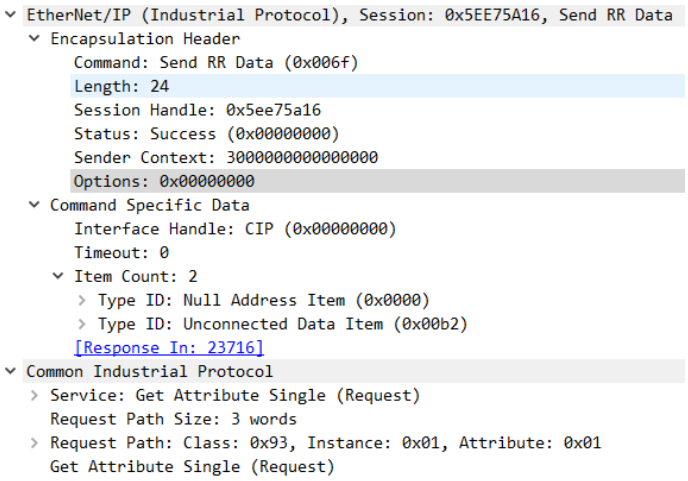
Obr. 5.21: Odpověď serveru s hodnotou 2 – vakuová přísavka.



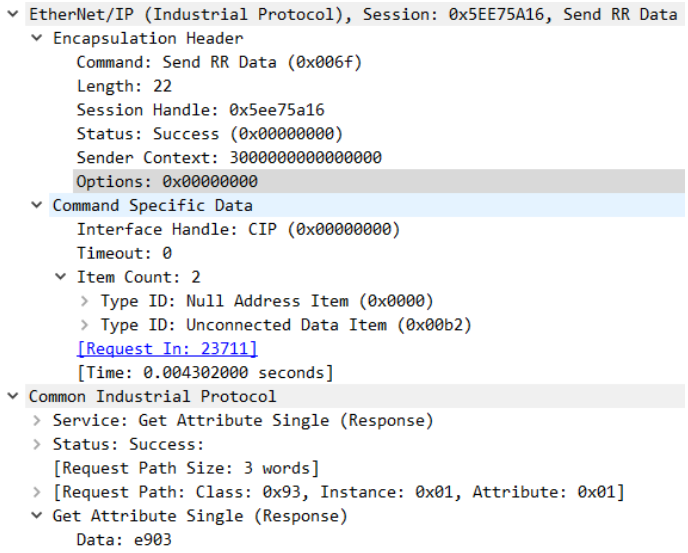
Obr. 5.22: Odpověď serveru s hodnotou 1 – zapnutí sání.

která nám udává, že se jedná právě o operaci „move“. Odpověď serveru si můžeme prohlédnout na obrázku 5.24.

Poté si klient postupně vyžádá hodnoty registru 0x02–0x07. Toto dělá vždy, ale k ovládání ostatních periferií s nimi nepracuje (jsou buď prázdné nebo obsahují minulé hodnoty do doby jejich přepisu novými).



Obr. 5.23: Žádost klienta o hodnotu registru 0x01.



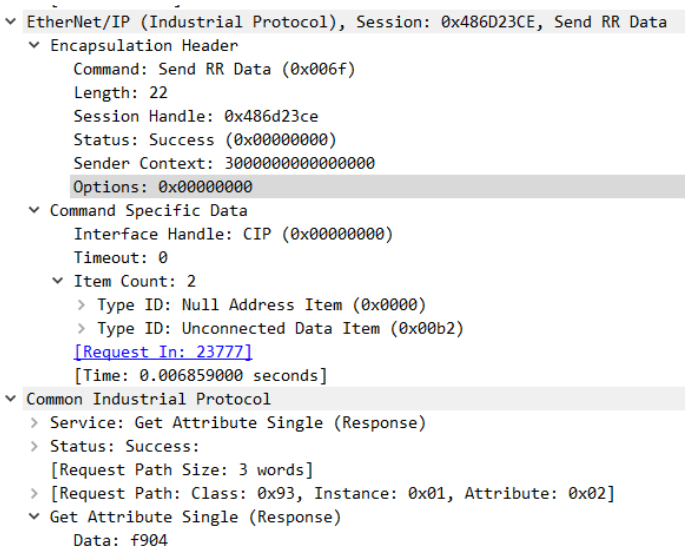
Obr. 5.24: Odpověď serveru s hodnotou 1 – operace „move“.

Pro operaci „move“ klient potřebuje registry 0x02–0x05, které reprezentují pohyb v následujícím sletu:

- **Registr 0x02:** Udává hodnotu pohybu v ose x.
- **Registr 0x03:** Udává hodnotu pohybu v ose y.
- **Registr 0x04:** Udává hodnotu pohybu v ose z.
- **Registr 0x05:** Udává hodnotu pohybu v ose r.

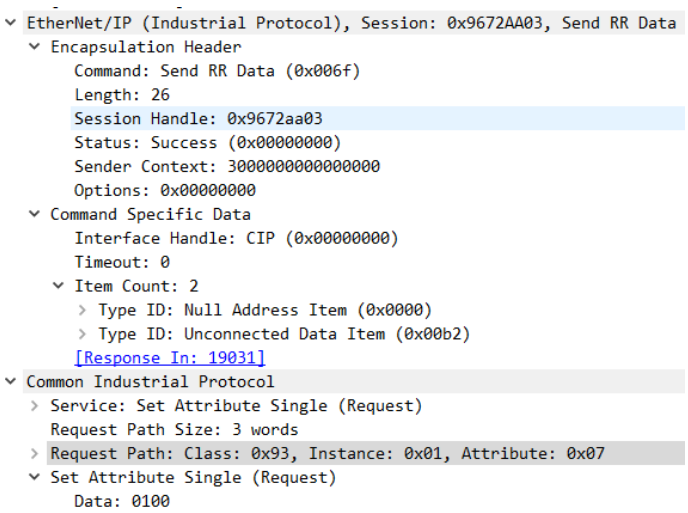
Na následujícím obrázku 5.25 můžeme vidět odpověď serveru s hodnotou registru

0x02, která nám udává pohyb v ose x. Přijatá hexadecimální hodnota je **f904**, kterou budeme opět číst jako **04f9**. Po převodu do dekadického tvaru se jedná o číslo **1273**, které nabyde po dekódování hodnoty **273**. Toto nám udává pohyb v ose x o 273 mm. Ostatní registry by nám udaly pohyb v osách [y, z, r] a postup by byl totožný.

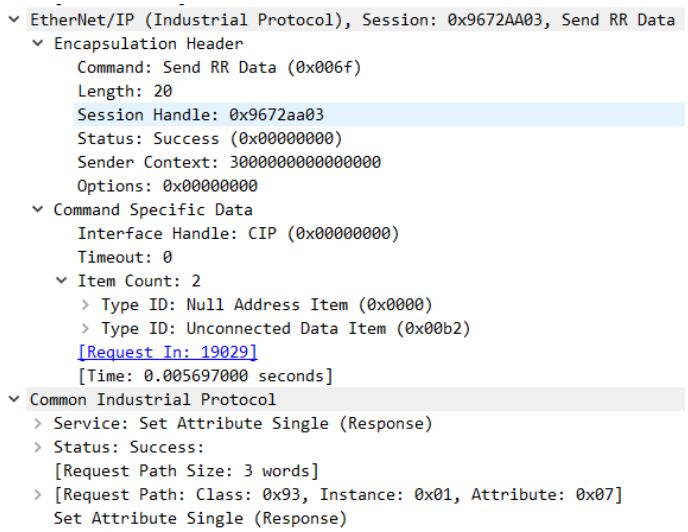


Obr. 5.25: Odpověď serveru s hodnotou pohybu v ose x.

Poté, co klient příjme parametry operace a danou operaci vykoná je na server do registru 0x07 zapsána hodnota **1** viz obrázek 5.26, která serveru dává informaci, že klient operaci dokončil a server může vysílat operaci další. Potvrzení přijetí zápisu registru můžeme vidět na obrázku 5.27. Jako u předchozích operací, jsou data převáděna z hexadecimální podoby do dekadické a dekódovány.



Obr. 5.26: Zápis hodnoty 1 do registru 0x07.



Obr. 5.27: Potvrzení přijetí zápisu ze strany serveru.

Závěr

Tato diplomová práce si kladla za cíl otestovat komunikaci na průmyslových protokolech a realizovat průmyslovou smyčku (zacyklený úkol), které umožní demonstrovat praktické řešení. Hlavním požadavkem bylo vytvořit funkční komunikaci na principu server-klient na bázi komunikace po průmyslových protokolech.

Teoretická část se věnuje rozsáhlé rešerši na téma seznámení se s chytrou továrnou a prolínání informačních technologií s průmyslovými. Podstatná část rešerše s ohledem na téma diplomové práce je věnována rozborem používaných průmyslových protokolů, které by bylo vhodné v dalším řešení implementovat. V první kapitole je popsána definice chytré továrny. Další kapitoly jsou věnovány konvergenci IT s OT včetně popisu používaných rozhraní a periferií. Předposlední kapitola je věnována průmyslovým protokolům včetně jejich vzniku, popisu a komunikace.

V praktické části jsme se seznámili s robotickou paží Dobot Magician, a za pomocí běžně dostupných komponent navrhli testovací prostředí. Dále byla detailně popsaná instalace serverové a klientské části včetně dostupných knihoven a problémy s tímto spjaté. Pro komunikaci byl implementován průmyslový protokol Modbus TCP a jeho funkcionalita byla předvedena na demonstrativním videu, které je přílohou této práce.

V další praktické části byl implementován průmyslový protokol EtherNET/IP a vytvoření funkčního celku – průmyslové smyčky. Průmyslová smyčka tvoří simulaci reálného provozu, která se cyklicky opakuje. Průmyslová smyčka byla vytvořena ze tří robotických paží Dobot Magician, posuvné kolejnice (rail), pásového dopravníku (Convyor belt) a senzoru barev (Color sensor). Průmyslový přepínač zajišťuje propojení více robotických paží Dobot Magician – pomocí tří mikropočítačů (Raspberry Pi 3B+).

Celý postup byl zdokumentován v poslední kapitole této diplomové práce. Komunikace dvou průmyslových protokolem byla analyzována programem Wireshark a finální ukázka implementace průmyslové smyčky byla demonstrována na videu, které je součástí příloh této práce.

Tímto bylo celé zadání splněno a představuje funkční celek, který má potenciál dalšího rozvoje. Pro další rozvoj práce by byla možnost implementace Dobot Vision Kitu (vizuální kit pro Dobota) s možností řízení operací Dobota v závislosti na detekci předmětů z obrazu. Stejně tak jako implementace dalších průmyslových protokolů a možnost průmyslovou smyčku vystavovat kybernetickým útokům.

Literatura

[1] *Smart Factories and the future of manufacturing* [online]. poslední aktualizace 26. 10. 2018 [cit. 19. 10. 2019]. Dostupné z URL: <https://www.automationmagazine.co.uk/articles/smart-factories-and-the-future-of-manufacturing/>.

[2] *The smart factory - Responsive, adaptive, connected manufacturing* [online]. poslední aktualizace 31. 8. 2017 [cit. 20. 10. 2019]. Dostupné z URL: <https://www2.deloitte.com/us/en/insights/focus/industry-4-0/smart-factory-connected-manufacturing.html#endnote-sup-6/>.

[3] *Germany Trade and Invest, Smart factory* [online]. poslední aktualizace 18. 8. 2017 [cit. 20. 10. 2019]. Dostupné z URL: <https://industrie4.0.gtai.de/INDUSTRIE40/Navigation/EN/Topics/Industrie-40/smart-factory.html/>.

[4] *The rise of the digital supply network* [online]. poslední aktualizace 1. 12. 2016 [cit. 21. 10. 2019]. Dostupné z URL: <https://www2.deloitte.com/us/en/insights/focus/industry-4-0/digital-transformation-in-supply-chain.html/>.

[5] *Information Technologies (IT) Vs Operational Technologies (OT)* [online]. poslední aktualizace 6. 3. 2019 [cit. 22. 10. 2019]. Dostupné z URL: <https://randed.com/information-technologies-it-vs-operational-technologies-ot/?lang=en/>.

[6] *IT vs. OT in Manufacturing: How Will Convergence Play Out?* [online]. poslední aktualizace 16. 1. 2014 [cit. 22. 10. 2019]. Dostupné z URL: <http://www.clresearch.com/research/detail.cfm?guid=8D3AB104-3048-79ED-99C3-8106D7556B6D/>.

[7] *Industrial Control Systems are ruling the world* [online]. poslední aktualizace 12. 3. 2015 [cit. 23. 10. 2019]. Dostupné z URL: <https://www.sentryo.net/industrial-control-systems-are-ruling-the-world/>.

[8] *What's the Difference Between OT, ICS, SCADA and DCS?* [online]. poslední aktualizace 1. 5. 2019 [cit. 23. 10. 2019]. Dostupné z URL: <https://www.sentryo.net/industrial-control-systems-are-ruling-the-world/>.

[9] *Distributed Control Systems (DCS) - What is DCS* [online]. poslední aktualizace 20. 8. 2019 [cit. 23. 10. 2019]. Dostupné z URL: <https://www.electricalinput.com/2018/08/distributed-control-systems-dcs-what-is.html/>.

[10] *PLC Working Principle with Industrial Applications* [online]. poslední aktualizace 29. 3. 2019 [cit. 24. 10. 2019]. Dostupné z URL: <https://www.watelectrical.com/industrial-applications-of-programmable-logic-controller/>.

[11] *Co by měl každý vědět o programovacích jazycích PLC* [online]. poslední aktualizace 29. 8. 2017 [cit. 25. 10. 2019]. Dostupné z URL: <https://www.elektroprumysl.cz/software/co-by-mel-kazdy-vedet-o-programovacich-jazycich-plc/>.

[12] *What are the Most Popular PLC Programming Languages?* [online]. poslední aktualizace 26. 11. 2018 [cit. 26. 10. 2019]. Dostupné z URL: <https://realpars.com/plc-programming-languages//>.

[13] *Human-machine interface* [online]. poslední aktualizace 12. 6. 2014 [cit. 27. 10. 2019]. Dostupné z URL: <https://www.britannica.com/technology/human-machine-interface/>.

[14] *What is the difference between SCADA and HMI?* [online]. poslední aktualizace 30. 7. 2018 [cit. 27. 10. 2019]. Dostupné z URL: <https://realpars.com/difference-between-scada-and-hmi/#at_pco=smlwn-1.0&at_si=5db435de44a239cc&at_ab=per-2&at_pos=0&at_tot=1/>.

[15] *What is RTU?* [online]. poslední aktualizace 3. 9. 2018 [cit. 28. 10. 2019]. Dostupné z URL: <https://realpars.com/rtu/>.

[16] *Introducing the MQTT Protocol - MQTT Essentials: Part 1* [online]. poslední aktualizace 12. 1. 2015 [cit. 29. 10. 2019]. Dostupné z URL: <https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/>.

[17] *MODBUS* [online]. poslední aktualizace 7. 7. 2004 [cit. 2. 11. 2019]. Dostupné z URL: <https://automatizace.hw.cz/clanek/2004070701/>.

[18] *The Modbus Protocol In-Depth* [online]. poslední aktualizace 17. 9. 2019 [cit. 3. 11. 2019]. Dostupné z URL: <https://www.ni.com/cs-cz/innovations/white-papers/14/the-modbus-protocol-in-depth.html/>.

[19] *Oficiální stránky standardu Modbus* [online]. © 2005 – 2019 [cit. 6. 11. 2019]. Dostupné z:<http://www.modbus.org/>.

[20] *Modbus ASCII VS Modbus RTU VS Modbus TCP/IP* [online]. poslední aktualizace 8. 10. 2017 [cit. 8. 11. 2019]. Dostupné z URL: <https://theautomization.com/modbus-ascii-vs-modbus-rtu-vs-modbus-tcpip/>.

[21] *MODBUS TCP/IP* [online]. © 2019 [cit. 6. 11. 2019]. Dostupné z URL :<https://www.rtautomation.com/technologies/modbus-tcpip/>.

[22] MACKAY, Steave. *Practical industrial data networks: design, installation and troubleshooting.*, Amsterdam: Newnes, 2004. s. 115–117. ISBN 9780750658072.

[23] *PROFINET – The Solution Platform for Process Automation* [online]. poslední aktualizace 2015 [cit. 11. 11. 2019]. Dostupné z URL: <https://www.profibus-profinet.cz/profinet/>.

[24] *PROFINET System Description* [online]. poslední aktualizace 2014 [cit. 11. 11. 2019]. Dostupné z URL: <https://www.profibus-profinet.cz/profinet/>.

[25] *PROFIsafe System DescriptionTechnology and Application* [online]. poslední aktualizace 2016 [cit. 13. 11. 2019]. Dostupné z URL: <https://www.profibus-profinet.cz/profinet/>.

[26] *What is Ethernet/IP?* [online]. poslední aktualizace 25. 2. 2019 [cit. 19. 11. 2019]. Dostupné z URL: <https://realpars.com/ethernet-ip/>.

[27] *EtherNet/IP: Implicit vs. Explicit Messaging* [online]. [cit. 19. 11. 2019]. Dostupné z URL: <https://library.automationdirect.com/ethernetip-implicit-vs-explicit-messaging/>.

[28] *Overview of DNP3 Protocol* [online]. [cit. 25. 11. 2019]. Dostupné z URL: <https://www.dnp.org/About/Overview-of-DNP3-Protocol/>.

[29] POPOVICI, Katalin a Pieter J. MOSTERMAN.: *Real-time simulation technologies: principles, methodologies, and applications*. Boca Raton, FL: CRC Press, 2013. ISBN 1439846650. s. 353–356.

[30] *Dobot Magician User Manual v1.7.0* [online]. poslední aktualizace 29. 9. 2019 [cit. 1. 12. 2019]. Dostupné z URL: <https://www.dobot.cc/downloadcenter/dobot-magician.html?sub_cat=73#sub-download/>.

[31] *Dobot Magician Interface description v2* [online]. poslední aktualizace 11. 7. 2019 [cit. 1. 12. 2019]. Dostupné z URL: <https://www.dobot.cc/downloadcenter/dobot-magician.html?sub_cat=73#sub-download/>.

[32] *Dobot Demo v2.0* [online]. poslední aktualizace 13. 3. 2019 [cit. 2. 12. 2019]. Dostupné z URL: <https://www.dobot.cc/downloadcenter/dobot-magician.html?sub_cat=72#sub-download/>.

[33] *pydobot 1.0.2* [online]. poslední aktualizace 22. 6. 2019 [cit. 2. 12. 2019]. Dostupné z URL: <https://pypi.org/project/pydobot/>.

[34] *Welcome to PyModbus's documentation!* [online]. [cit. 2. 12. 2019]. Dostupné z URL: <https://pymodbus.readthedocs.io/en/latest/>.

[35] *Welcome to cpppo's documentation! 4.0.6.* [online]. [cit. 3. 12. 2019]. Dostupné z URL: <https://pypi.org/project/cpppo/>.

[36] *Raspberry Pi 3B+ Specs and Benchmarks* [online]. [cit. 3. 12. 2019]. Dostupné z URL: <https://magpi.raspberrypi.org/articles/raspberry-pi-3bplus-specs-benchmarks/>.

[37] *Sliding Rail Kit* [online]. poslední aktualizace 29. 9. 2019 [cit. 1. 2. 2020]. Dostupné z URL: <https://www.dobot.cc/products/sliding-rail-kit-overview.html>.

[38] *Sliding Rail User Guide v1* [online]. poslední aktualizace 11. 9. 2018 [cit. 1. 2. 2020]. Dostupné z URL: <https://download.dobot.cc/product-manual/sliding-rail-kit/pdf/en/Sliding-Rail-User-Guide.pdf>.

[39] *Conveyor Belt Kit* [online]. poslední aktualizace 29. 9. 2019 [cit. 2. 2. 2020]. Dostupné z URL: <https://www.dobot.cc/products/conveyor-belt-kit-overview.html>.

[40] *Conveyor Belt User Manual* [online]. poslední aktualizace 26. 3. 2019 [cit. 2. 2. 2020]. Dostupné z URL: <https://download.dobot.cc/product-manual/conveyor-belt-kit/User-manual/pdf/en/Conveyor-Belt-User-Guide.pdf>.

[41] *Raspbian Buster with desktop and recommended software* poslední aktualizace 13. 2. 2020 [online]. [cit. 4. 2. 2020]. Dostupné z URL: <https://downloads.raspberrypi.org/raspbian_full_latest>.

[42] *Win32 Disk Imager* poslední aktualizace 27. 11. 2019 [online]. [cit. 7. 2. 2020]. Dostupné z URL: <https://win32diskimager.download/Win32DiskImager-1.0.0-src.zip>.

[43] *Dobot Communication Protocol v1.1.5* poslední aktualizace 23. 4. 2020 [online]. [cit. 25. 5. 2020]. Dostupné z URL: <https://download.dobot.cc/product-manual/dobot-magician/pdf/en/Dobot-Communication-Protocol-V1.1.5.pdf>.

Seznam symbolů, veličin a zkratek

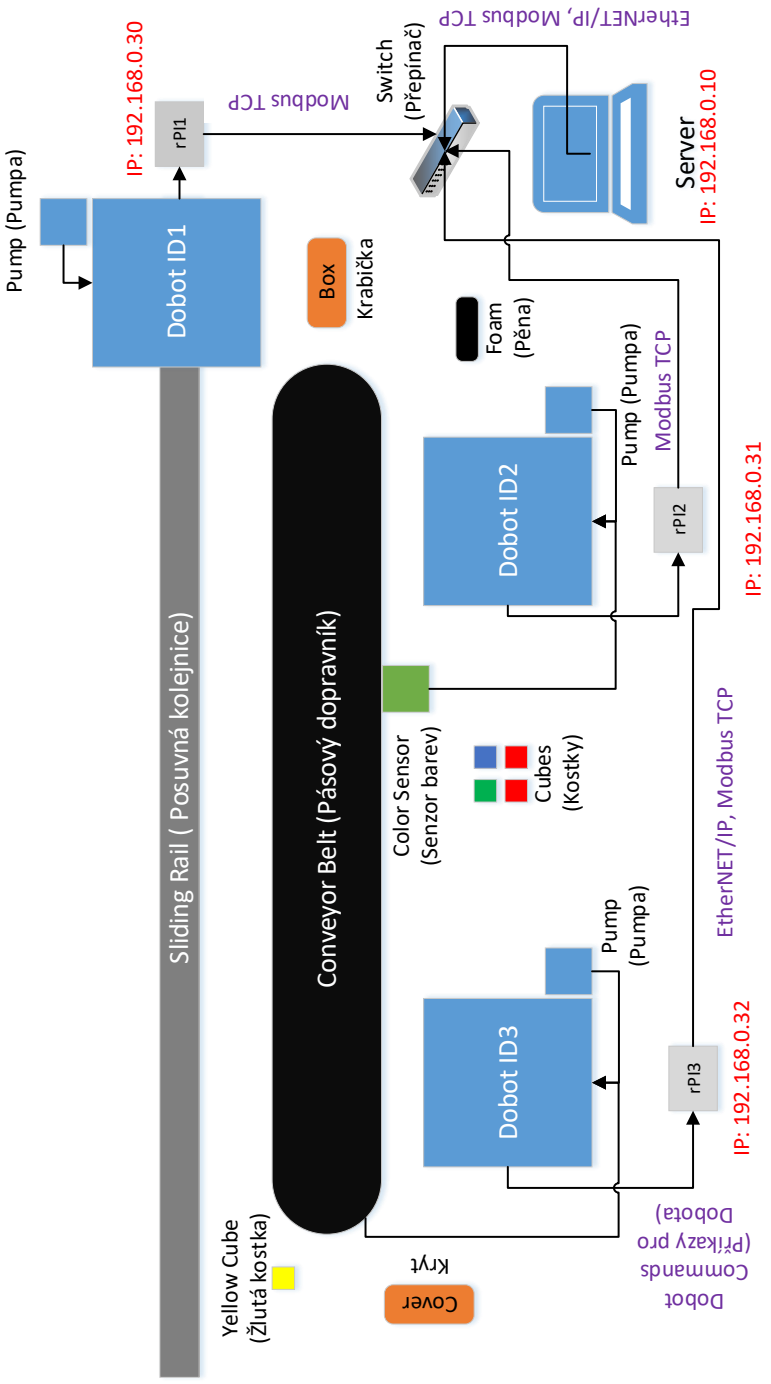
AI	Umělá inteligence – Artifical Intelligence
ADU	Aplikační datová jednotka – Application Data Unit
ARM	Zdokonalený počítač typu RISC – Advanced RISC Machine
BT	„modrozub“ – bezdrátová technologie připojení periferií v nelicencovaném pásmu 2.4 GHz – BlueTooth
CAT5e	Kabel kategorie 5e – Category 5e cable
CIP	Průmyslový protokol pro aplikace průmyslové automatizace – Common Industrial Protocol
CRC	Cyklický redundantní součet – Cyclic redundancy check
CRM	Řízení vztahu se zákazníky – Customer Relationship Managment
DCOM	Je objektově orientovaný mechanismus, který strukturuje způsob,jakým může klient vyhledávat, požadovat a přijímat data ze serveru – Distributed Component Object Model
DCP	Jedná se o protokol fungující na linkové vrstvě pro konfiguraci názvů stanic a IP adres – Discovery and basic Confguration Protokol
DCS	Distribuovaný kontrolní systém – Distributed Control System
DNP3	Je sada komunikačních protokolů používaných mezi komponenty v systémech automatizace procesů – Distributed Network Protocol 3
ENIP	EtherNET/IP protokol – EtherNET/IP protocol
ERP	Plánování podnikových zdrojů – Enterprise Resource Planning
FBD	Jazyk funkčního blokového schématu – Function Block Diagram
FCS	Kontrola rámce – Frame check sequence
FDL	Datové spojení Fieldbus – Fieldbus Data Link
FTP	Protokol pro přenos souborů – Simple Mail Transfer Protocol
GSD	Hlavní popis stanice tj. konfigurace, moduly. – General Station Description
HDLC	Vysokoúrovňové řízení datového spoje – High-Level Data Link Control
HMI	Rozhraní mezi člověkem a strojem – Human-Machine Interface
HTTP	Internetový protokol pro komunikaci s WWW servery – Hypertext Transfer Protocol
I/O	Vstup/Výstup – Input/output
ICS	Řídicí systém – Industrial control system
ID	Identifikace – IDentification
IEA	Mezinárodní energetická agentura – International Energy Agency
IED	Inteligentní elektronická zařízení – Intelligent electronic devices
IEEE	Mezinárodní nezisková profesní organizace usilující o vzestup

	technologie související s elektrotechnikou – Institute of Electrical and Electronics Engineers
IL	Jazyk seznamu instrukcí – Instruction List
IP	Protokol internetu – Internet Protocol
IRT	Isochronií přenos v reálném čase – Isochronous Real-Time
ISO	Mezinárodní organizace pro normalizaci – International Organization for Standardization
IT	Informační technologie – Information technology
LD	Jazyk příčkového diagramu – Ladder diagram
LED	Světlo emitující dioda – Light Emitting Diode
LLC	Podvrstva linkové vrstvy – Logical Link Control
MAC	MAC adresa – Media Acess Control
NRT	Přenos, který neprobíhá v reálném čase – Non-Real-Time
ODVA	Asociace dodavatelů zařízení DeviceNet – Open DeviceNet Vendors Association
OT	Provozní technologie – Operating technology
PCD	Popis komponent PROFINETu – PROFINET Component Description
PDU	Datová jednotka protokolu – Protocol Data Unit
PLC	Programovatelný logický automat – Programmable Logic Controller
RDP	Síťový protokol, který umožňuje uživateli ovládat vzdálený počítač prostřednictvím počítačové sítě – Remote Desktop Protocol
RPC	Technologie dovolující programu zpracovat kód na jiném místě, než je umístění volajícího programu – Remote Procedure Call
rPI	Raspberry Pi – Raspberry Pi
RT	Přenos v reálném čase – Real Time
RTC	Hodiny reálného času – Real Time Clock
RTU	Zařízení pro vzdálené řízení – Remote Terminal Unit
SCADA	Dispečerské řízení a sběr dat – Supervisory Control And Data Acquisition
SDRAM	Synchronní DRAM – Synchronous DRAM
SFC	Sekvenční funkční diagram – Sequential Function Charts
SIL	Bezpečnostní úroveň integrity – Safety Integrity Level
SMTP	Internetový protokol elektronické pošty – Simple Mail Transfer Protocol
SNMP	Protokol sloužící pro správu sítě – Simple Network Management Protocol
SQL	Standardizovaný strukturovaný dotazovací jazyk – Strucured Query Language

SSH	Zabezpečený protokol a zároveň komunikační program – Secure Shell
SSL	Protokol/vrstva, která běží mezi transportní a aplikační vrstvou a poskytuje zabezpečení komunikace pomocí šifrování a autentizace komunikujících stran – Secure Sockets Layer
ST	Jazyk strukturovaného textu – Structured Text
TCP	Základní protokol sady internet na transportní vrstvě – Transmission Control Protocol
UART	Univerzální asynchronní přijímač-vysílač – Universal asynchronous receiver-transmitter
UTP	Nestíněna kroucená dvojlinka – Unshielded Twisted Pair
USB	Univerzální sériová sběrnice – Universal Serial Bus
UDP	Základní protokol sady internet na transportní vrstvě – User Datagram Protocol
WWW	Světová komunikační síť – World Wide Web
XML	Rozšiřitelný značkovací jazyk – Extensible Markup Language
xRDP	Síťový protokol, který umožňuje uživateli ovládat vzdálený počítač prostřednictvím počítačové sítě (open source) – xRemote Desktop Protocol

Seznam příloh

A Návrh průmyslové smyčky



Obr. A.1: Schéma návrhu průmyslové smyčky.

B Knihovna dobot.py

```
def _set_ptp_withL_cmd(self, x, y, z, r, l, mode, wait):
        wait = self.wait
        msg = Message()
        msg.id = 86
        msg.ctrl = 3
        msg.params = bytearray([])
        msg.params.extend(bytearray([mode]))
        msg.params.extend(bytearray(struct.pack('f', x)))
        msg.params.extend(bytearray(struct.pack('f', y)))
        msg.params.extend(bytearray(struct.pack('f', z)))
        msg.params.extend(bytearray(struct.pack('f', r)))
        msg.params.extend(bytearray(struct.pack('f', l)))
        return self._send_command(msg, wait)

    def move_to_withL(self, x, y, z, r, l, wait=True):
        wait = self.wait
        return self._set_ptp_withL_cmd(x, y, z, r, l, mode=0x02, wait=wait)

    def move_conveyor(self, distance, direction, motor=0, speed=6000, wait=False):
        # distance in cm
        # direction: 0=forward, 1=backward
        wait = False
        if direction == 1:
            speed = speed * -1
        self.start_stepper(speed, motor, wait)  # cca 5cm/sec
        self.set_wait(228 * distance)
        self.stop_stepper(motor, wait)
        return 1

    def enable_rail(self, wait=False):
        wait = self.wait
        msg = Message()
        msg.id = 3
        msg.ctrl = 3
        msg.params = bytearray([])
        msg.params.extend(bytearray(struct.pack('i', int(True))))
        return self._send_command(msg, wait)

    def _set_color_sensor(self, state, port=0x01, wait=False):
        wait = self.wait
        msg = Message()
        msg.id = 137
        msg.ctrl = 0x03
        msg.params = bytearray([])
        msg.params.extend(bytearray([int(state)]))
        msg.params.extend(bytearray([port]))
        msg.params.extend(bytearray([1]))
        return self._send_command(msg, True)

    def enable_color_sensor(self, port=0x01, wait=False):
        wait = self.wait
        return self._set_color_sensor(1, port, wait)

    def disable_color_sensor(self, port=0x01, wait=False):
        wait = self.wait
        return self._set_color_sensor(0, port, wait)


    def read_color(self, port=0x01, wait=False):
        wait = False
        msg = Message()
        msg.id = 137
        msg.ctrl = 0
        msg.params = bytearray([])
        msg.params.extend(bytearray(struct.pack('i', port)))
        response = self._send_command(msg, wait)
        r = struct.unpack_from('?', response.params, 0)[0]
        g = struct.unpack_from('?', response.params, 1)[0]
        b = struct.unpack_from('?', response.params, 2)[0]
        return [r, g, b]

    def _set_emotor(self, index, enabled, speed, wait=False):
        wait = False
        msg = Message()
        msg.id = 135
        msg.ctrl = 0x03
```

```
        msg.params = bytearray(struct.pack('B', index))
        msg.params.extend(bytearray(struct.pack('B', enabled)))
        msg.params.extend(bytearray(struct.pack('i', speed)))
        return self._send_command(msg, wait=wait)

    def _set_wait_cmd(self, ms, wait):
        wait = True
        msg = Message()
        msg.id = 110
        msg.ctrl = 0x03
        msg.params = bytearray(struct.pack('I', ms))
        return self._send_command(msg, wait=wait)

    def home(self, wait=False):
        wait = self.wait
        msg = Message()
        msg.id = 31
        msg.ctrl = 0x03
        return self._send_command(msg, True)



class CommunicationProtocolIDs():
    GET_SET_DEVICE_SN = 0
    GET_SET_DEVICE_NAME = 1
    GET_POSE = 10
    RESET_POSE = 11
    GET_ALARMS_STATE = 20
    CLEAR_ALL_ALARMS_STATE = 21
    SET_GET_HOME_PARAMS = 30
    SET_HOME_CMD = 31
    SET_GET_HHTTRIG_MODE = 40
    SET_GET_HHTTRIG_OUTPUT_ENABLED = 41
    GET_HHTTRIG_OUTPUT = 42
    SET_GET_ARM_ORIENTATION = 50
    SET_GET_END_EFFECTOR_PARAMS = 60
    SET_GET_END_EFFECTOR_LAZER = 61
    SET_GET_END_EFFECTOR_SUCTION_CUP = 62
    SET_GET_END_EFFECTOR_GRIPPER = 63
    SET_GET_JOG_JOINT_PARAMS = 70
    SET_GET_JOG_COORDINATE_PARAMS = 71
    SET_GET_JOG_COMMON_PARAMS = 72
    SET_GET_PTP_JOINT_PARAMS = 80
    SET_GET_PTP_COORDINATEP_ARAMS = 81
    SET_GET_PTP_JUMP_PARAMS = 82
    SET_GET_PTP_COMMON_PARAMS = 83
    SET_PTP_CMD = 84
    SET_CP_CMD = 91
    SET_QUEUED_CMD_START_EXEC = 240
    SET_QUEUED_CMD_STOP_EXEC = 241
    SET_QUEUED_CMD_CLEAR = 245
    GET_QUEUED_CMD_CURRENT_INDEX = 246
```

C Zdrojový kód server.py

```
import copy
import threading
import time
from random import randint

import logging

from classes.Arm1 import Arm1
from classes.Arm2 import Arm2
from classes.Arm3 import Arm3
import server_config as config
from classes.ModbusServer import ModbusServer
from classes.CommandQueue import CommandQueue

from classes.EnipContext import EnipContext

actual_cube = []
cubes_stack = []
color = [0, 0, 0]
arm1 = Arm1()
arm2 = Arm2()
arm3 = Arm3()
armsModbus = [arm1, arm2]
armsEnip = [arm3]
queue = CommandQueue()


def start_server():
    server = ModbusServer(config.slaves, log=log)
    modbus = threading.Thread(target=server.run, args=(config.server_address, config.port,))
    modbus.start()
    enip_server = EnipContext()
    update_context(server, enip_server)


def fill_queue():
    global cubes_stack, color, actual_cube
    actual_cube = []
    cubes_stack = copy.deepcopy(arm2.positions.cubes)
    queue.add(arm1.move_box_to_belt())
    queue.add(arm2.insert_foam())
    queue.add(arm3.move_box_to_color_sensor())
    queue.add([conditional_action])
    queue.add(arm3.move_belt_to_cover())
    queue.add(arm1.put_yellow_cube_into_box())
    queue.add(arm3.put_cover())
    queue.add(arm3.remove_cover())


def all_operations_done(server_instance, enip_instance):
    result = 1
    for arm in armsModbus:
        result = result * server_instance.is_function_finished(arm.id)
    for arm in armsEnip:
        result = result * enip_instance.is_function_finished(arm.id)
    return bool(result)


def update_context(server_instance, enip_context):
    while True:
        while not all_operations_done(server_instance, enip_context):
            time.sleep(config.refresh_time)
            log.debug("waiting until no client is busy")
        update_color(server_instance)
        item = queue.get_next()
        if not item:
            log.debug("items empty, filling")
            fill_queue()
            reset_color(server_instance)
            item = queue.get_next()
        log.info("sending operation %s with params %s to slaveId %s"
        % (item.operation, item.coordinates, item.slave_id))
        if armsEnip[0].id == item.slave_id:
            enip_context.update_context(item.opncoordinates)
            enip_context.reset_function_finished(item.slave_id)
```

```
        server_instance.update_context(item.slave_id, item.opncoordinates)
        server_instance.reset_function_finished(item.slave_id)
        time.sleep(config.refresh_time)


def _set_logging():
    global log
    logging.basicConfig(filename="server.log", filemode="w", format='
␣␣␣␣%(asctime)s␣-␣%(levelname)s␣-␣%(message)s')
    log = logging.getLogger(__name__)

    console = logging.StreamHandler()
    console.setLevel(logging.DEBUG)
    log.addHandler(console)

    log.setLevel(logging.DEBUG)


def disassembly():
    return [
        arm1.positions.yellow_cube_in_box, arm1.set_suction_cup(1), arm1.positions.yellow_cube,
        arm1.set_suction_cup(0),
        arm1.positions.optimal,
        arm3.move_conveyor(-40),
        arm2.positions.optimal, arm2.positions.cube_in_box, arm2.set_suction_cup(1),
        arm2.positions.return_cube,
        arm2.set_suction_cup(0),
        arm3.move_conveyor(-18),
        arm2.positions.optimal, arm2.positions.box, arm2.set_suction_cup(1), arm2.positions.foam,
        arm2.set_suction_cup(0),
        arm1.positions.rail_min, arm1.positions.box_on_belt, arm1.set_suction_cup(1),
        arm1.positions.box,
        arm1.set_suction_cup(0),
        arm1.positions.optimal, arm2.positions.optimal, arm3.positions.optimal
    ]


def update_color(server_instance):
    global color
    color = server_instance.get_color(2)
    return color


def reset_color(server_instance):
    global color
    color = [0, 0, 0]
    server_instance.reset_color(2)


def is_green():
    return color[1] == 1


def conditional_action():
    global actual_cube, cubes_stack
    if is_green():
        positions = [arm2.positions.cube_in_box, arm2.set_suction_cup(0), arm2.positions.optimal]
        arm2.positions.return_cube = actual_cube
        queue.add(disassembly())
    else:
        cube_index = randint(0, len(cubes_stack) - 1)
        cube = cubes_stack.pop(cube_index)
        log.debug("next␣round␣-\t␣cube:␣%s␣\t␣cube-pos:␣%s:␣\t␣in␣cubes-count:␣%s" % (
        cube_index + 1, str(cube.raw), len(cubes_stack) + 1))
        positions = [cube, arm2.set_suction_cup(1), arm2.positions.color_sensor, arm2.get_color(),
        conditional_action]
        if (actual_cube):
            positions.insert(0, arm2.set_suction_cup(0))
            positions.insert(0, actual_cube)
        actual_cube = cube
    return positions


conditional_action.name = "conditional_action"

if __name__ == "__main__":
    _set_logging()
    fill_queue()
    start_server()
```

D Zdrojový kód client.py

```
import logging
import time

import clientConfig as config
from pydobot import Dobot
from pymodbus.client.sync import ModbusTcpClient as ModbusClient
from serial.tools import list_ports

from enumerators.actions import Actions, ActionsRegisters
from enumerators.registers import Registers


class Client(object):
    def __init__(self):
        self.log = self._set_logging()
        self.client = ModbusClient(config.address, port=config.port)
        self.client.connect()
        self.previous_parameters = []
        self.dobot = self._set_dobot()
        if(config.has_rail):
            self.dobot.enable_rail()
        self.main_loop()

    def main_loop(self):
        while True:
            try:
                self.read_and_move()
            except Exception as e:

                self.log.debug("nothing␣received" + str(e))
            finally:
                self.operation_finished()
                time.sleep(config.wait_time)

    def read_and_move(self):
        operation = self._read_operation()
        parameters = self._read_operation_parameters(operation)
        self.decode(parameters)
        if self.is_same_as_previous(parameters):
            self.log.debug("same␣data,␣doing␣nothing")
        else:
            self._do_operation(operation, parameters)
        self.previous_parameters = parameters

    def operation_finished(self):
        try:
            self.client.write_registers(Registers.function_finished, 1, unit=config.slave_id)
              # function finished
        except Exception as e:
            self.log.debug("no␣connection" + str(e))

    def is_same_as_previous(self, parameters):
        return self.previous_parameters == parameters

    def _set_logging(self):
        logging.basicConfig(filename="client"+str(config.slave_id)+".log", filemode="w", format='
␣␣␣␣␣␣␣␣%(asctime)s␣-␣%(levelname)s␣-␣%(message)s')
        log = logging.getLogger(__name__)
        console = logging.StreamHandler()
        console.setLevel(logging.DEBUG)
        log.addHandler(console)
        log.setLevel(logging.DEBUG)
        return log

    def _set_dobot(self):
        port = list_ports.comports()[0].device
        device = Dobot(port=port, verbose=False)
        return device

    def _read_operation(self):
        return self._read_registers(Registers.coordinates, 1)[0]

    def _read_registers(self, address, count):
        result = self.client.read_holding_registers(address, count, unit=config.slave_id)
        result_register = []
```

```
        for i in range(count):
            result_register.append(result.registers[i])
        return result_register

    def _read_operation_parameters(self, operation):
        params = self._read_registers(Registers.coordinates,
        ActionsRegisters.registers[operation])
        params.pop(0)
        return params

    def _do_operation(self, operation, operation_parameters):
        self.log.debug("operation: %s with parameters: %s" % (operation, operation_parameters))
        if operation == int(Actions.HOME):
            self.dobot.home()
        elif operation == int(Actions.MOVE):
            if config.safe_move:
                x, y, z, r, l = operation_parameters
                xorig, yorig, zorig, rorig, _, _, _, _ = self.dobot.pose()
                self.dobot.move_to(xorig, yorig, zorig + config.safeZOffset, rorig,
                wait=config.run_synchronously)
                self.dobot.move_to(x, y, z + config.safeZOffset, r, wait=config.run_synchronously)
            if config.has_rail:
                self.dobot.move_to_withL(x, y, z, r, l, wait=config.run_synchronously)
            else:
                self.dobot.move_to(x, y, z, r, wait=config.run_synchronously)
        elif operation == int(Actions.SUCK):
            print(int(operation_parameters[0]))
            if(operation_parameters[0] == 1):
                state = True
            else:
                state = False
            self.dobot.suck(state)
        elif operation == int(Actions.MOVE_CONVEYOR):
            direction = 0 if operation_parameters[0] < 0 else 1
            self.dobot.move_conveyor(abs(operation_parameters[0]), direction)
        elif operation == int(Actions.READ_COLOR):
            self.dobot.enable_color_sensor()
            time.sleep(1)
            self.client.write_registers(Registers.color,self.dobot.read_color(),
            unit=config.slave_id)
            self.dobot.disable_color_sensor()

    def decode(self,item):
        for index in range(0, len(item)):
            item[index] -= 1000


if __name__ == "__main__":
    client = Client()
```

E Zdrojový kód client_ethip.py

```
import copy
import logging
import time

import clientConfig as config
from pydobot import Dobot
from cpppo.server.enip.get_attribute import proxy_simple, attribute_operations
from cpppo.server.enip import client

import cpppo

logging.basicConfig(**cpppo.log_cfg)

from serial.tools import list_ports

from enumerators.actions import Actions, ActionsRegisters
from enumerators.registers import Registers


class Client(object):
    def __init__(self):
        self.log = self._set_logging()
        self.previous_parameters = []
        self.via = dobotdevice(host=config.address, port="44818", timeout=1)

        self.dobot = self._set_dobot()
        self.params = ["op","x", "y", "z", "r", "l", "opfinished"]

        if(config.has_rail):
            self.dobot.enable_rail()
        self.main_loop()
    def process(self,par, val):
            self.process.values[par] = val

    process.done = False
    process.values = {}
    def main_loop(self):
        self.operation_finished()
        while True:

            try:
                self.read_and_move()
            except Exception as e:
                raise
                print("nothing received" + str(e))
            finally:
                time.sleep(config.wait_time)


    def read_and_move(self):
        operation = self._read_operation()
        parameters = self._read_operation_parameters(operation)
        if self.is_same_as_previous(parameters):
            self.log.debug("same data, doing nothing")
        else:
            self._do_operation(operation, parameters)
            self.operation_finished()
        self.previous_parameters = parameters

    def operation_finished(self):
        try:
            param = 'opfinished = (INT)%s' % (1)
            with self.via:
                val, = self.via.write(
                self.via.parameter_substitution(param), checking=True)
            print("op finished")
        except Exception as e:
            self.log.debug("no connection" + str(e))

    def is_same_as_previous(self, parameters):
        return self.previous_parameters == parameters

    def _set_logging(self):
        logging.basicConfig(filename="client"+str(config.slave_id)+".log", filemode="w", format='
%(asctime)s - %(levelname)s - %(message)s')
        log = logging.getLogger(__name__)
```

```
        console = logging.StreamHandler()
        console.setLevel(logging.DEBUG)
        log.addHandler(console)
        log.setLevel(logging.DEBUG)
        return log

    def _set_dobot(self):
        port = list_ports.comports()[0].device
        device = Dobot(port=port, verbose=False)
        return device

    def _read_operation(self):
        registers = copy.deepcopy(self._read_registers()[0])
        return registers
    def _decode(self,item):
        arr = copy.deepcopy(item)
        for index in range(0, len(arr)):
            arr[index] -= 1000
        return arr

    def _read_registers(self):
        tags = ['@0x93/␣␣1/1', '@0x93/␣␣1/2', '@0x93/␣␣1/3', '@0x93/␣␣1/4',
        '@0x93/␣␣1/5', '@0x93/␣␣1/6']
        op=list(proxy_simple("192.168.0.10").read([('@0x93/␣␣1/1', 'INT')]))[0][0]

        x=list(proxy_simple("192.168.0.10").read([('@0x93/␣␣1/2', 'INT')]))[0][0]
        y=list(proxy_simple("192.168.0.10").read([('@0x93/␣␣1/3', 'INT')]))[0][0]
        z=list(proxy_simple("192.168.0.10").read([('@0x93/␣␣1/4', 'INT')]))[0][0]
        r=list(proxy_simple("192.168.0.10").read([('@0x93/␣␣1/5', 'INT')]))[0][0]
        l=list(proxy_simple("192.168.0.10").read([('@0x93/␣␣1/6', 'INT')]))[0][0]

        params = [op,x,y,z,r,l]
        print(params)
        return self._decode(params)


    def _read_operation_parameters(self, operation):
        params = copy.deepcopy(self._read_registers())
        params.pop(0)
        return params

    def _do_operation(self, operation, operation_parameters):
        print("operation:␣%s␣with␣parameters:␣%s" % (operation, operation_parameters))
        if operation == int(Actions.HOME):
            self.dobot.home()
        elif operation == int(Actions.MOVE):
            if config.safe_move:
                x, y, z, r, l = operation_parameters
                xorig, yorig, zorig, rorig, _, _, _, _ = self.dobot.pose()
                self.dobot.move_to(xorig, yorig, zorig + config.safeZOffset, rorig,
                wait=config.run_synchronously)
                self.dobot.move_to(x, y, z + config.safeZOffset, r, wait=config.run_synchronously)
            if config.has_rail:
                self.dobot.move_to_withL(x, y, z, r, l, wait=config.run_synchronously)
            else:
                self.dobot.move_to(x, y, z, r, wait=config.run_synchronously)
        elif operation == int(Actions.SUCK):
            print(int(operation_parameters[0]))
            state = bool(operation_parameters[0])
            self.dobot.suck(state)
        elif operation == int(Actions.MOVE_CONVEYOR):
            direction = 0 if operation_parameters[0] < 0 else 1
            self.dobot.move_conveyor(abs(operation_parameters[0]), direction)
        elif operation == int(Actions.READ_COLOR):
            self.dobot.enable_color_sensor()
            time.sleep(1)
            self.client.write_registers(Registers.color,self.dobot.read_color(),
            unit=config.slave_id)
            self.dobot.disable_color_sensor()

class dobot(proxy_simple):
    pass
class dobotdevice(dobot):
    PARAMETERS = dict(dobot.PARAMETERS,
                      op=dobot.parameter('@0x93/␣␣1/1', 'INT', ''),

                      x=dobot.parameter('@0x93/␣␣1/2', 'INT', 'mm'),
                      y=dobot.parameter('@0x93/␣␣1/3', 'INT', 'mm'),
                      z=dobot.parameter('@0x93/␣␣1/4', 'INT', 'mm'),
                      r=dobot.parameter('@0x93/␣␣1/5', 'INT', 'mm'),
```

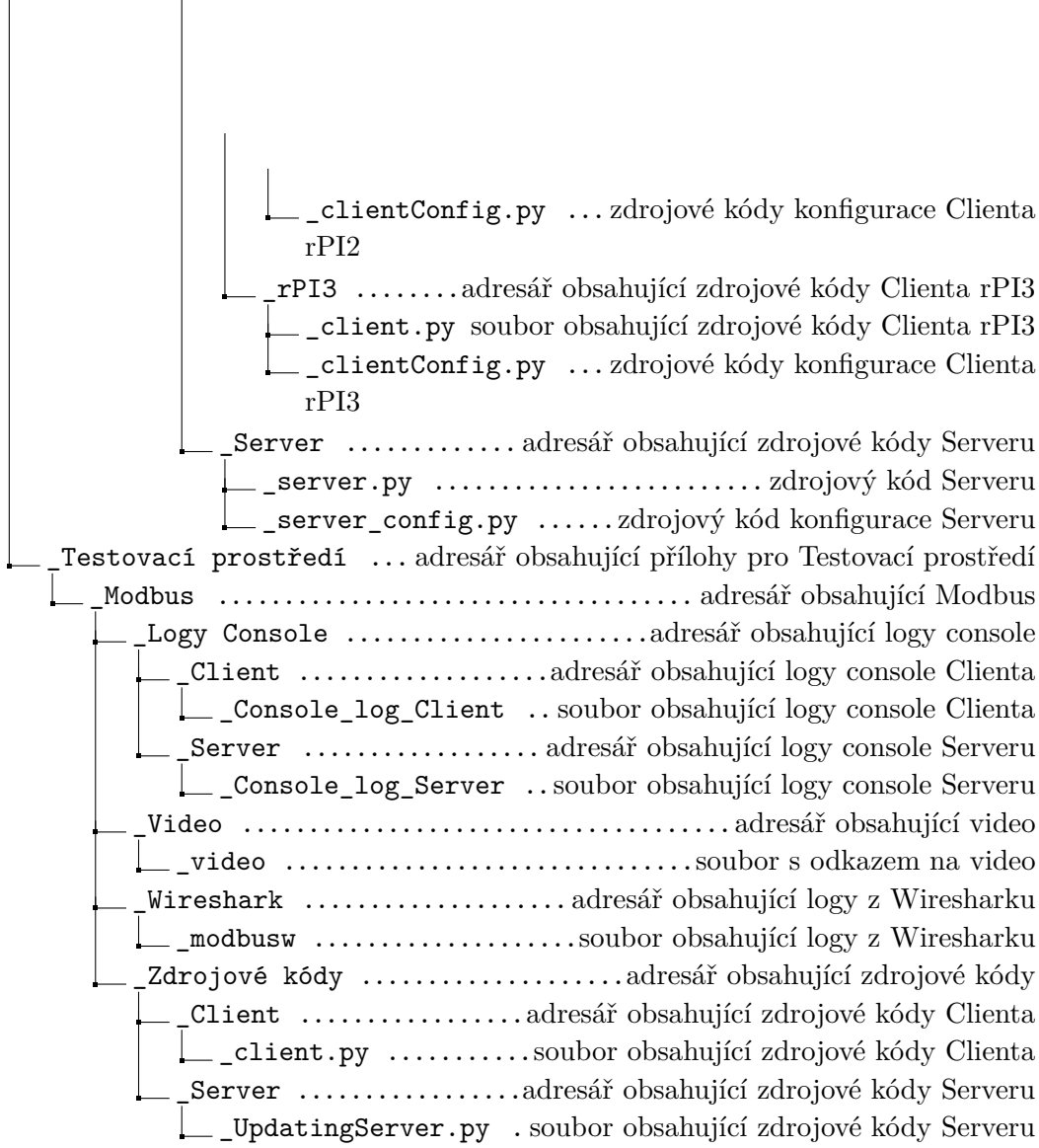
```
                        l=dobot.parameter('@0x93/␣␣1/6', 'INT', 'mm'),
                        opfinished=dobot.parameter('@0x93/␣␣1/7', 'INT', 'mm'),

                        )

def failure(exc):
    failure.string.append(str(exc))
if __name__ == "__main__":
    client = Client()
```

F Obsah přiloženého CD

- `/` ..... kořenový adresář přiloženého DVD
  - `_Kohoutek_diplomova_prace.pdf` ..... diplomová práce v PDF
  - `_Přílohy` ..... adresář obsahující přílohy
    - `_Průmyslová smyčka` .... adresář obsahující přílohy pro Průmyslovou smyčku
      - `_Logy Console` ..... adresář obsahující logy z console
        - `_Client` ..... adresář obsahující logy z console Clienta
          - `_rpi1-terminal` ..... soubor obsahující logy z console Clienta rPI1
          - `_rpi2-terminal` ..... soubor obsahující logy z console Clienta rPI2
          - `_rpi3-terminal` ..... soubor obsahující logy z console Clienta rPI3
        - `_Server` ..... adresář obsahující logy z console Serveru
          - `_server-terminal` ..... soubor obsahující logy z console Serveru
      - `_Video` ..... adresář obsahující video
        - `_video` ..... soubor obsahující odkaz na video
      - `_Wireshark` ..... adresář obsahující logy z Wiresharku
        - `_dobotwireshark` ..... soubor obsahující logy z Wiresharku
      - `_Zdrojové kody` ..... adresář obsahující zdrojové kódy
        - `_Knihovna` ..... adresář obsahující zdrojové kódy knihovny
          - `_dobot.py` ..... soubor obsahující zdrojové kódy knihovny
        - `_Knihovna` ..... adresář obsahující zdrojové kódy knihovny
          - `_ModbusxEthernetIP` ... adresář zdrojových kódů Modbus x EthIP
            - `_Client` ..... adresář obsahující zdrojové kódy Clientů
              - `_rPI1` ..... adresář obsahující zdrojové kódy Clienta rPI1
                - `_client.py` soubor obsahující zdrojové kódy Clienta rPI1
                - `_clientConfig.py` ... zdrojové kódy konfigurace Clienta rPI1
              - `_rPI2` ..... adresář obsahující zdrojové kódy Clienta rPI2
                - `_client.py` soubor obsahující zdrojové kódy Clienta rPI2
                - `_clientConfig.py` ... zdrojové kódy konfigurace Clienta rPI2
              - `_rPI3` ..... adresář obsahující zdrojové kódy Clienta rPI3
                - `_client.py` soubor obsahující zdrojové kódy Clienta rPI3
                - `_clientConfig.py` ... zdrojové kódy konfigurace Clienta rPI3
            - `_Server` ..... adresář obsahující zdrojové kódy Serveru
              - `_server.py` ..... zdrojový kód Serveru
              - `_server_config.py` ..... zdrojový kód konfigurace Serveru
          - `_OnlyModbus` ..... adresář zdrojových kódů Modbus
            - `_Client` ..... adresář obsahující zdrojové kódy Clientů
              - `_rPI1` ..... adresář obsahující zdrojové kódy Clienta rPI1
                - `_client.py` soubor obsahující zdrojové kódy Clienta rPI1
                - `_clientConfig.py` ... zdrojové kódy konfigurace Clienta rPI1
              - `_rPI2` ..... adresář obsahující zdrojové kódy Clienta rPI2
                - `_client.py` soubor obsahující zdrojové kódy Clienta rPI2


_clientConfig.py ... zdrojové kódy konfigurace Clienta rPI2
_rPI3 ..... adresář obsahující zdrojové kódy Clienta rPI3
_client.py soubor obsahující zdrojové kódy Clienta rPI3
_clientConfig.py ... zdrojové kódy konfigurace Clienta rPI3
_Server ..... adresář obsahující zdrojové kódy Serveru
_server.py ..... zdrojový kód Serveru
_server_config.py ..... zdrojový kód konfigurace Serveru
_Testovací prostředí ... adresář obsahující přílohy pro Testovací prostředí
_Modbus ..... adresář obsahující Modbus
_Logy Console ..... adresář obsahující logy console
_Client ..... adresář obsahující logy console Clienta
_Console_log_Client .. soubor obsahující logy console Clienta
_Server ..... adresář obsahující logy console Serveru
_Console_log_Server .. soubor obsahující logy console Serveru
_Video ..... adresář obsahující video
_video ..... soubor s odkazem na video
_Wireshark ..... adresář obsahující logy z Wiresharku
_modbusw ..... soubor obsahující logy z Wiresharku
_Zdrojové kódy ..... adresář obsahující zdrojové kódy
_Client ..... adresář obsahující zdrojové kódy Clienta
_client.py ..... soubor obsahující zdrojové kódy Clienta
_Server ..... adresář obsahující zdrojové kódy Serveru
_UpdatingServer.py . soubor obsahující zdrojové kódy Serveru