VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2020

Bc. Marek Magáth

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

DEPARTMENT OF CONTROL AND INSTRUMENTATION

KOMUNIKAČNÍ ROZHRANÍ PRO TESTBED I4.0

COMMUNICATION INTERFACE FOR INDUSTRY 4.0 TESTBED

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Marek Magáth
AUTHOR

VEDOUCÍ PRÁCE Ing. Václav Kaczmarczyk, Ph.D.
SUPERVISOR

BRNO 2020

Diplomová práce

magisterský navazující studijní obor **Kybernetika, automatizace a měření**

Ústav automatizace a měřicí techniky

Student: Bc. Marek Magáth ***ID:*** 186135

Ročník: 2 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Komunikační rozhraní pro testbed I4.0

POKYNY PRO VYPRACOVÁNÍ:

Navrhněte programovou strukturu a implementujte komunikační rozhraní, které bude použitelné jako vstupní brána pro poskytování informací a řízení autonomních výrobních buněk testbedu Industry 4.0.

1. Seznamte se s testbedem průmyslu 4.0 a s jeho funkcí
2. Definujte požadavky na komunikační rozhraní umožňující distribuované řízení a zvolte vhodný komunikační protokol.
3. Navrhněte strukturu univerzálního programového modulu pro komunikační rozhraní procesních buněk tesbedu.
4. Navrhněte strukturu komunikačního programového modulu výrobku.
5. Implementujte programový modul komunikačního rozhraní procesních buněk.
6. Implementujte komunikační programový modul výrobku.
7. Řešení otestujte a vyhodnoťte dosažené výsledky.

DOPORUČENÁ LITERATURA:

OPC-UA specifikace komunikačních profilů

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

Vedoucí práce: Ing. Václav Kaczmarczyk, Ph.D.

doc. Ing. Václav Jirsík, CSc.
předseda oborové rady

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Táto diplomová práca sa zaoberá vytvorením všeobecného komunikačného rozhrania medzi inteligentným výrobkom a procesnými bunkami v rámci *test bedu Barman*. Cieľom je vytvoriť distribuovanú výrobu podľa koceptu *Industry 4.0*, kde si každý výrobok riadi svoju výrobu sám. V prvej časti sa práca venuje analýze *test bedu*, kde sú rozobrané požiadavky a spôsob výroby. Ďalej výberom komunikačného protokolu, ktorý by spĺňal požiadavky *Industry 4.0* a distribuovaného riadenia. Výsledkom rešerše bol komunikačný protokol *OPC UA*. Pomocou tohto protokolu, budú spolu komunikovať výrobok a procesná bunka. K zvolenému protokolu som vybral vhodnú implementáciu, ktorá sa použije pre tvorbu komunikačných modulov výrobku a procesnej bunky. Dáta o výrobe budú uložené v *RFID* čipe, ktorý sa nachádza vo výrobku. Časť práce sa venuje práve tvorbe tejto štruktúry dát, ktorá obsahuje receptúru, ale aj stavové dáta o výrobe. Tvorba *RFID* čítačky je práca iného študenta, s ktorým som riešil spôsob výmeny dát medzi obslužnou aplikáciou čítačky a komunikačným modulom výrobku. Pri tvorbe komunikačného modulu procesnej bunky som riešil návrh informačného modelu, ktorý popisuje celú procesnú bunku a dynamické vytváranie uzlov v adresovom priestore. Uviedol som niekoľko príkladov, ako som pracoval s vybranou implementáciou *OPC UA* a ako som ju použil v oboch aplikáciách. V rámci riešenia som implementoval decentralizované vyhľadávanie procesných buniek v sieti. Túto vlastnosť bolo jednoduché implementovať s *OPC UA*, konkrétne so servisom `LDS-ME`. Pri komunikačnom module výrobku som riešil ovládanie výroby a spôsob komunikácie s procesnou bunkou. A v poslednej časti sa venujem manuálu na správne zostavenie projektov, konkrétne na operačnom systéme *Linux* a opisujem spôsob testovania celého riešenia spolu s výsledkami.

KĽÚČOVÉ SLOVÁ

OPC UA, Industry 4.0, AAS, C++, Open62541, Distribuovaná výroba, test bed, CMake, MQTT

ABSTRACT

This diploma thesis deals with the creation of a general communication interface between the smart product and process cells within *test bed Barman*. The aim is to create distributed production according to the *Industry 4.0* concept, where each product manages it's production itself. The first part deals with the analysis of *test bed*, where are discussed requirements and method of production. Then with the selection of a communication protocol that would meet *Industry 4.0* and distributed control requirements. The result of the research was *OPC UA* communication protocol. Using this protocol, the product and the process cell will communicate with each other. I chose the appropriate implementation for the selected protocol to be used for creating communication modules of the product and the process cell. Production data will be stored in an *RFID* chip located in the product. Part of the work is devoted to creation this data structure, which contains the recipe as well as production state data. Creation of *RFID* reader is the work of another student, with whom I solved the method of data exchange between application of the reader and the communication module of the product. When creating a communication module of the process cell, I solved the design of an information model that describes the entire process cell, and the dynamic creation of nodes in the address space. I have mentioned a few examples of how I worked with a selected implementation of *OPC UA* and how I used it in both applications. I implemented decentralized search of another process cells in the local network. This feature was easy to implement with *OPC UA*, specifically with the *LDS-ME* service. I solved the control of production for the communication module of the product and method of communication with the process cell. And in the last part I deal with manual, which describes how to compile projects correctly, specifically on the *Linux* operating system, and I have described a way to test the whole solution together with the results.

KEYWORDS

OPC UA, Industry 4.0, AAS, C++, Open62541, Distributed manufacturing, test bed, CMake, MQTT

MAGÁTH, Marek. *Komunikační rozhraní pro testbed I4.0*. Brno, 2020, 93 s. Diplomová práca. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedúci práce: Ing. Václav Kaczmarczyk, Ph.D.

Vysázeno pomocí balíčku thesis verze 3.03; http://latex.feec.vutbr.cz

VYHLÁSENIE

Vyhlasujem, že som svoju diplomovú prácu na tému „Komunikační rozhraní pro testbed I4.0“ vypracoval samostatne pod vedením vedúceho diplomovej práce, využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej diplomovej práce ďalej vyhlasujem, že v súvislosti s vytvorením tejto diplomovej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona Českej republiky č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákoníka Českej republiky č. 40/2009 Sb.

Brno 1.6.2020 ....

podpis autora

PODĎAKOVANIE

Rád by som poďakoval vedúcemu diplomovej práce pánovi Ing. Václavovi Kaczmarczykovi, Ph.D., ale tak isto aj Ing. Ondrejovi Baštánovi a Ing. Tomášovi Benešlovi za odborné vedenie, konzultácie, trpezlivosť a podnetné návrhy k práci.

Brno 1.6.2020 ....

podpis autora

Obsah

Úvod **14**

1 Test bed I4.0 - Barman **15**

1.1 Test bed . . . 15
1.2 Konštrukcia . . . 15
1.2.1 Konštrukcia procesných buniek . . . 16
1.3 Sieť a riadiaci program . . . 17
1.4 Popis jednotlivých procesných buniek . . . 18
1.4.1 Sklad pohárov . . . 18
1.4.2 Sklad alkoholu . . . 19
1.4.3 Drvič ľadu . . . 20
1.4.4 Sódovač . . . 21
1.4.5 Shaker . . . 22
1.4.6 Robotické rameno *SCARA* . . . 23
1.5 Proces výroby nápoja . . . 24
1.6 Všeobecné zhrnutie . . . 24
1.7 Požiadavky na distribuované riadenie . . . 25

2 Výber komunikačného protokolu **27**

2.1 Výber implementácie *OPC UA* . . . 29

3 Návrh informačného modelu bunky **32**

3.1 Všeobecný postup pri vytváraní informačných modelov . . . 32
3.2 Informačný model procesnej bunky . . . 33

4 Receptúra **39**

4.1 *API* obslužnej aplikácie *RFID* čítačky . . . 39
4.2 Model receptúry . . . 42

5 Aplikácia **45**

5.1 Štruktúra riešenia . . . 45
5.2 Knižnica `open62541-cpp-api` . . . 49
5.2.1 Nastavenie knižnice *Open62541* . . . 49
5.2.2 Štruktúra projektu . . . 52
5.2.3 Trieda `UaNodeContext` . . . 52
5.2.4 Pridávanie uzlov v adresovom priestore . . . 60
5.3 Komunikačný modul procesnej bunky - `process-cell` . . . 61
5.3.1 Konfiguračný súbor . . . 61

5.3.2 Spôsob komunikácie a mapovania dát z *PLC* do komunikačného modulu bunky . . . 63
5.3.3 Mapovanie *OPC UA* metódy a spúšťanie výroby pomocou `RunAction` metódy . . . 67
5.3.4 Decentralizované vyhľadávanie procesných buniek v sieti . . . 69
5.4 Komunikačný modul produktu - `product` . . . 70
5.4.1 Konfiguračný súbor . . . 70
5.4.2 Stavový automat výroby . . . 71
5.4.3 Komunikácia s aplikáciou `process-cell` . . . 72
5.4.4 Filtrovanie procesných buniek v sieti . . . 72
5.4.5 Spracovanie notifikácii . . . 73
5.5 Zapisovanie receptúry do *RFID* - `recepture-writer` . . . 73

6 Nasadenie aplikácií a testovanie **75**
6.1 Zostavenie aplikácie zo zdrojových súborov . . . 76
6.2 Testovanie distribuovanej výroby . . . 77

7 Záver **80**

Literatúra **83**

Zoznam symbolov, veličín a skratiek **85**

Zoznam príloh **86**

A Knižnica `open62541-cpp-api` **87**

B Aplikácia `process-cell` **88**

C Aplikácia `product` **90**

D Aplikácia `recepture-writer` **92**

E Obsah priloženého CD **93**

Zoznam obrázkov

1.1 Test bed *Barman* [9]. 16
1.2 Bloková schéma zariadení v jednej procesnej bunke. 17
1.3 Procesná bunka - `Sklad pohárov` [6]. 18
1.4 Procesná bunka - `Sklad alkoholu` [7]. 19
1.5 Procesná bunka - `Drvič ľadu` [8]. 20
1.6 Procesná bunka - `Sódovač` [9]. 21
1.7 Procesná bunka - `Shaker` [10]. 22
1.8 Transportná bunka - `Robotické rameno` *`SCARA`* [11]. 23
1.9 Bloková schéma rozloženia procesných buniek. 25
1.10 Rozdiel medzi typmi riadenia výroby [3]. 26
2.1 Referenčný 3D model *RAMI 4.0* [5]. 28
2.2 Využitie *OPC UA* v *RAMI 4.0* v komunikačnej a reprezentačnej vrstve [4]. 29
2.3 Bloková schéma zariadení a komunikácií medzi aplikáciami v rámci jednej bunky. 30
3.1 Grafické symboly [12]. 33
3.2 Postup pri vytváraní informačného modelu. 34
3.3 Informačný model procesnej bunky. 34
3.4 Informačný model dátového typu info a manažment. 35
3.5 Informačný model dátového typu výroby a rezervácie. 38
4.1 Štruktúra pamäte čipu *NTAG213* [19]. 39
4.2 *UML* diagram receptúry. 44
5.1 Štruktúra aplikácií. 46
5.2 *UML* diagram rozhrania `ICommunicaiton`. 64
5.3 *UML* diagram rozhrania `ICompositeBuilder`. 65
5.4 Kompozit návrhový vzor. 66
5.5 Štruktúra generickej metódy na spúšťanie výroby. 67
5.6 Stavový automat výroby procesnej bunky. 68
5.7 Stavový diagram aplikácie. 71
5.8 Postup aplikácie `recepture-writer`. 74
6.1 Úvodná obrazovka *Nano Pi*. 75
6.2 Bloková schéma zariadení pri testovaní. 78
6.3 Adresový priestor procesnej bunky `BaverageCell` zobrazený pomocou grafického *OPC UA* klienta počas testovania. 79
A.1 *UML* diagram `UaNodeContext`. 87
B.1 Funkčný blok spracovania *OPC UA* metódy. 89
C.1 Priebeh aplikácie `product`. 90

C.2 *UML* diagram triedy `ProcessCellClient` 91

Zoznam tabuliek

4.1 Hlavička receptúry 42
4.2 Kusovník receptúry 43
4.3 Procedúra receptúry 43
4.4 Stavové dáta výroby 43
B.1 Dáta blok exportovaný do formátu *csv*. 88

Zoznam výpisov

4.1 Formát *Json* správy s prečítanými dátami z *RFID* čipu. 40
4.2 Formát *Json* správy na zápis nových dát do *RFID* čipu. 41
4.3 Formát *Json* správy so statusom zápisu sektoru a číslom sektoru. . . 41
5.1 Ukážka agregovacieho `CMakelists.txt` 47
5.2 Ukážka `CMakelists.txt` z projektu `process-cell` 48
5.3 Ukážka nastavení *Open62541* z `CMakelists.txt` 51
5.4 Štruktúra spätného volania `UA_DataSource` [16]. 53
5.5 Štruktúra spätného volania `UA_ValueCallback` [16]. 53
5.6 Spätné volanie `UA_MethodCallback` [16]. 53
5.7 Štruktúra `UA_NodeTypeLifecycle` [16]. 54
5.8 Metóda `UA_Server_setNodeContext` pre nastavenie kontextu [16]. . . 55
5.9 Hlavička funkcie pridávania uzla premennej [16]. 60
5.10 Konfiguračný súbor aplikácie procesnej bunky. 61
5.12 Konfiguračný súbor aplikácie výrobku. 70
5.13 Vyhľadávanie servera pomocou schopností. 72
5.14 Spracovanie asynchronných požiadavkov a odpovedí [16]. 73
D.1 Receptúra v *Json* formáte. 92

Úvod

Táto diplomová práca sa zaoberá veľmi rozsiahlej téme, vytvorením distribuovaného riadenia v priemyselnej výrobe podľa konceptu *Industry 4.0*. Hlavným cieľom *Industry 4.0* je zvýšenie kyber bezpečnosti a efektívnosť, ktorá znižuje výrobné náklady a tým pádom aj cenu za jeden produkt. Produkčné systémy by sa mali stať viac flexibilné s cieľom ponúkať výrobky na mieru, a tým splniť požiadavky zákazníkov. Z toho vyplýva produkcia v menších počtoch, čo zvyšuje komplexnosť a cenu výroby. Tento koncept sa nezameriava len na vertikálnu integráciu, ale aj na horizontálnu, kde sa produkt sleduje od návrhu k objednávke, cez výrobu, až po odoslanie zákazníkovi, zbieranie pracovných dát a recykláciu (post-produkčný životný cyklus) [1]. Na zvládnutie tejto komplexnosti sa decentralizuje štruktúra výroby. Doposiaľ neboli vytvorené presné postupy, návody, štandardy na vytvorenie takejto výroby. Snaha bude definovať komunikačné moduly výrobnej bunky a výrobku, tak aby ich bolo možné použiť už v stávajúcich výrobných strojoch, ako aj pri vytváraní nových strojov.

Riadenie sa aplikuje na už vytvorený *test bed* s názvom *Barman*, ktorý vytvorili študenti bakalárskeho štúdia ako svoje záverečné práce. *Barman* predstavuje mini továreň určenú na výrobu miešaných nápojov podľa objednávok prichádzajúcich z nadradeného *ERP* systému.

Treba zdôrazniť, že celý *test bed* vznikol s myšlienkou vytvoriť mini továreň za účelom testovania rôznych prístupov riadenia výroby podľa konceptu *Industry 4.0*. Celý tento výrobný koncept má už vymyslenú kostru ľuďmi, ktorí sa podieľali na jeho vytvorení, hlavne z jeho konštrukčnej, hardvérovej, softvérovej stránky a približným princípom, ako by výroba mohla prebiehať.

Keďže ide o distribuované riadenie, predstava je taká, že každý výrobok si bude riadiť svoju výrobu sám. Bunky su koncipované, tak aby poskytovali jeden typ služby s rôznymi parametrami, ale nevylučuje sa, že výrobná bunka môže poskytovať aj viac ako jednu službu. V našom prípade sa jedná o služby ako nalievanie, miešanie, pridanie ľadu, transport a ďalšie.

Na spodnej časti pohára sa bude nachádzať *RFID* čip, kde je možné uložiť určité množstvo dát. V časti výrobnej bunky, kde sa prijíma pohár, bude umiestená *RFID* čítačka, ktorá bude obsluhovaná embedded zariadením. Zhotovením tejto čítačky spolu s výberom embedded zariadenia a obslužnej aplikácie sa zaoberá kolega *Bc. Kubíček* v rámci jeho diplomovej práce. Z toho vyplýva ďalšia úloha, a to dohodnúť spôsob výmeny dát medzi obslužnou aplikáciou čítačky a komunikačným modulom výrobku, ktorý bude tieto dáta potrebovať. Podrobnejší popis *test bedu* bude stručne popísaný v prvej kapitole aby sa čitateľ zoznámil s jeho koncepciou.

Mojou úlohou je oboznámiť sa s týmto novým konceptom výroby a vymyslieť

riadenie v rámci možností, ktoré ponúka *test bed Barman*. Distribuované riadenie bude prebiehať medzi produktami a výrobnými bunkami, ktoré ponúkajú parametrizovateľné služby. Budem sa zaoberať vytvorením štandardných dátových štruktúr, ktoré budú vedieť popísať čo najviac typov výrobných buniek a produktov. Tým sa zabezpečí využitie aj v iných aplikáciách.

Ďalej sa bude práca zaoberať rešeršou a výberom vhodného komunikačného protokolu použitého na riadenie v aplikácii *Industry 4.0*. Na základe vybraného protokolu spravím ďalšiu rešerš na dostupné knižnice. Budem dbať nato, aby bol tento protokol otvorený a knižnica bola *OpenSource* s dobrou dokumentáciou. Taktiež, aby bola knižnica spustiteľná na embedded zariadeniach.

Následne vytvorím aplikácie, komunikačný modul bunky a výrobku. Bunka by mala pomocou tohto modulu ponúkať dáta a služby iným zariadeniam vo výrobe. Výrobok by mal tieto služby využívať a dostať sa do finálneho stavu.

Na konci práce zhrniem dosiahnuté výsledky a podelím sa o skúsenosti a názory s vývojom tohto riešenia.

Pri všetkých týchto bodoch sa budem držať platných špecifikácii *Industry 4.0*, ktoré boli vydané do konca roku 2019.

1 Test bed I4.0 - Barman

V tejto úvodnej kapitole sa zoznámime s *test bedom*, či už z konštrukčnej, hardvérovej alebo softvérovej stránky, aby sme vedeli určiť, čo máme k dispozícii a limity pri návrhu komunikačného rozhrania, či už pre procesnú bunku alebo výrobok. Všetko, čo bude v tejto kapitole spomenuté, mi bolo predané buď slovne, alebo som sa o tom dočítal z bakalárskych práci študentov, ktorí vytvárali procesné bunky. Ako som spomenul v úvode, celý návrh fungovania *test bedu* je už navrhnutý podľa konceptu *Industry 4.0*, mojou úlohou je implementovať komunikačné rozhrania pre výrobné bunky a výrobok tak, aby si výrobok riadil výrobu sám za seba.

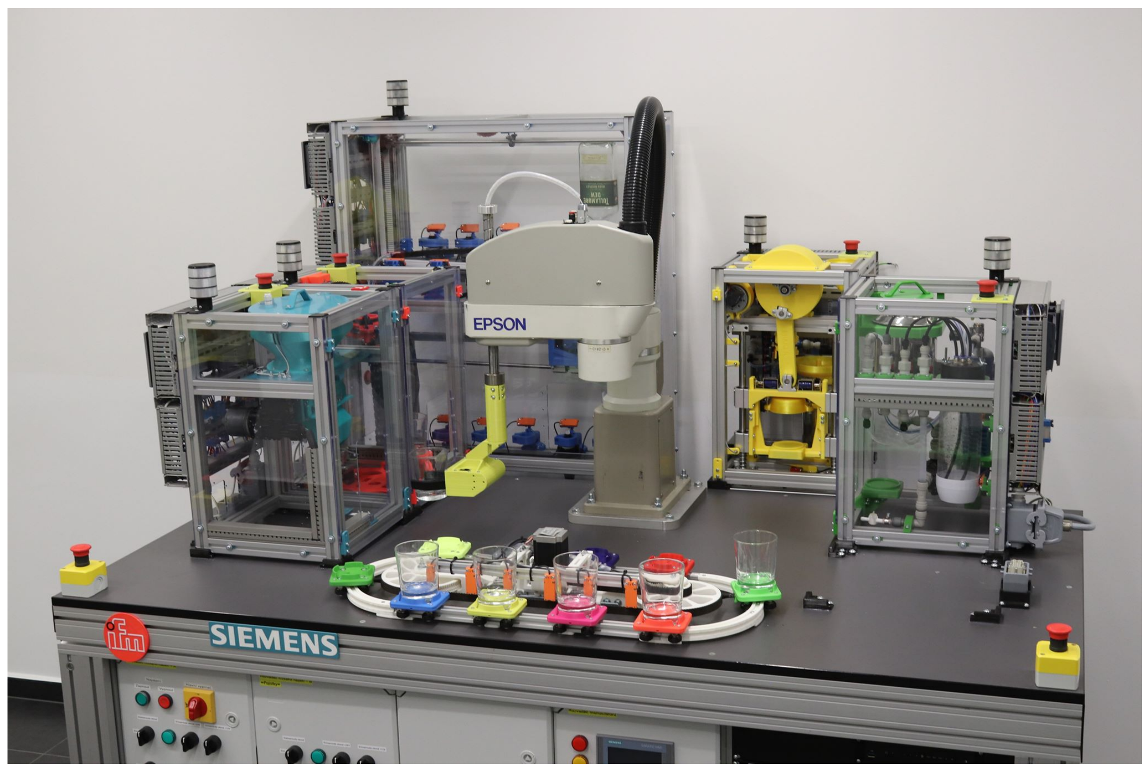
1.1 Test bed

Cieľom *test bedu* s názvom *Barman* je prakticky demonštrovať a testovať rôzne prístupy ku konceptu *Industry 4.0*. Jedná sa o mini továreň, ktorej úlohou je pripravovať nápoje na základe plne voliteľných objednávok zákazníka z nadradeného *ERP* systému. *Test bed* je navrhnutý podľa konceptu *Industry 4.0* ako maximálne modulárny. Skladá sa z produktov, čo predstavuje pohár, a každý element výroby, ktorý sa podieľa na zhotovení produktu sa nazýva bunka. Názov bunka vychádza z toho, že dané zariadenie vykonáva len jednu činnosť pre rôzne vstupné parametre a disponuje vlastným riadiacim systémom. Takáto štruktúra nám umožňuje vytvárať vysoko škálovateľné produkty. Jedna bunka predstavuje *CPPS*, v preklade kyberfyzikálny produkčný systém. Prepojenie niekoľkých *CPPS* vytvára takzvaný *kompozitný systém*. Tento formát ponúka jednoduchú rozšíriteľnosť a ľahkú výmenu jednotlivých autonómnych buniek. Stačí iba bunky vymeniť na svojich pozíciách, alebo nahradiť bunku s bunkou, ktorá spĺňa inú funkciu. To v praxi znamená drastickú redukciu času a práce potrebných k prestavbe továrne pri zmene požiadaviek na výrobu. Tieto bunky sú popísané podrobnejšie nižšie.

1.2 Konštrukcia

Celú mini továreň predstavuje stôl zhotovený z hliníkových rámov o rozmere 2x1x1 m. Nachádzajú sa tu dve podlažia. Horná plocha slúži na umiestnenie štyroch menších výrobných buniek na presne vyznačené miesta. Medzi týmito bunkami sa nachádza transportná bunka s dopravníkom. Ďalej sú tu umiestnené stop tlačítka na krajoch stola. Pod touto hlavnou plochou sa nachádza druhá plocha, ktorá obsahuje podporujúce zariadenia ako sieťové prvky, hlavný rozvádzač a iné podporné

zariadenia, ktoré by obsahovala normálna továreň. Väčšia bunka, ktorá má vlastnú konštrukciu sa nachádza za týmto stolom a je pevne prichytená k stolu.



Obr. 1.1: Test bed *Barman* [9].

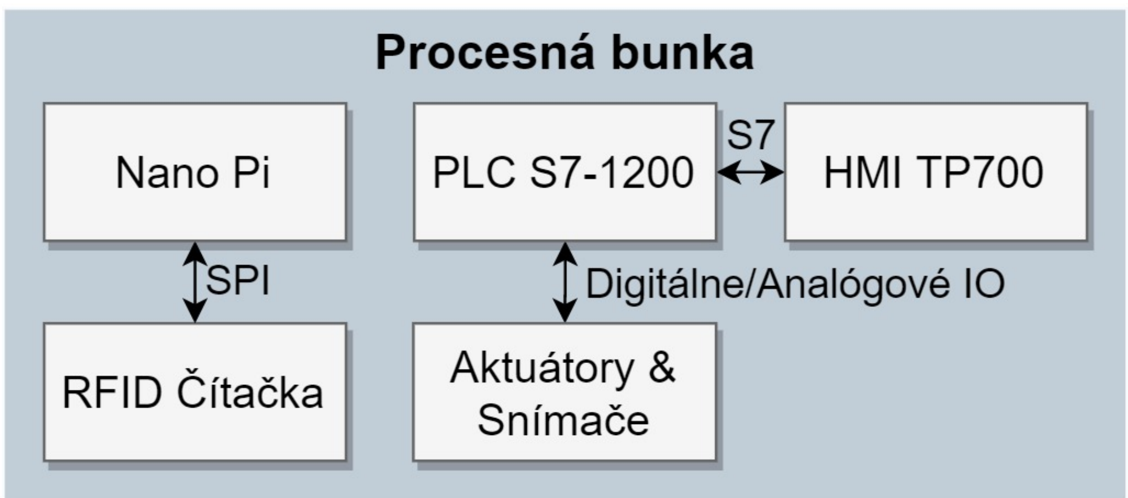
1.2.1 Konštrukcia procesných buniek

Každá výrobná bunka je zhotovená z hliníkových rámov, menšie bunky, ktoré sú umiestnené na stole majú rozmer 330x330x500 mm a väčšia bunka umiestnená za stolom má rozmer 1600x760x330 mm. Každá bunka má niekoľko bezpečnostných prvkov a je opláštená plexisklom kvôli tomu, aby spĺňala požiadavky na bezpečnosť.

Jediný otvor je pre vstup pohára. V tomto vstupe sa nachádza podnos pre pohár spolu s *RFID* čítačkou umiestnenou na spodnej strane s čítacím smerom hore. Táto čítačka je pripojená k *Nano Pi*, na ktorom beží obslužná aplikácia pre túto čítačku. Táto aplikácia poskytuje informáciu o prítomnosti pohára a nespracované dáta v bytoch vyčítané z čipu bez formátu. Každá bunka má svoj elektrický rozvádzač, kde je privedený elektrický prívod a sieťový prívod. Ďalej sa tu nachádza riadiaci systém *PLC S7-1200*, ktorý riadi jednotlivé funkcie danej bunky. Taktiež tu je dotykový panel *TP-700*, z ktorého je možné sledovať aktuálny stav bunky a procesné dáta alebo si môže operátor bunku prepnúť do ručného ovládania.

Všetky poháre disponujú *RFID* čipom pripevneným na spodnej časti. Niektoré bunky majú mechanizmus, pomocou ktorého si pohár zo vstupu vyberajú a premiestňujú v rámci nej na miesto, kde aplikujú danú operáciu, čiže pohár stratí kontakt s *RFID* čítačkou.

Na nasledujúcom obrázku je zobrazená bloková schéma zariadení s naznačeným tokom dát a použitými komunikačnými protokolmi.



Obr. 1.2: Bloková schéma zariadení v jednej procesnej bunke.

1.3 Sieť a riadiaci program

Všetky bunky sa nachádzajú v jednej lokálnej sieti a zariadenia majú dopredu určené a nastavené *IP* adresy. V rámci jednej bunky sú do siete pripojené *PLC*, ktoré riadi výrobný proces v bunke, dotyková obrazovka, ktorá slúži na sledovanie aktuálneho stavu bunky a procesných dát. Ako posledné je tu *Nano Pi*, na ktorom beží obslužná aplikácia pre *RFID* čítačku.

Riadiaci program je napísaný podľa štandardu *S88*. Využíva hlavne členenie programu podľa fyzického modelu. *PLC* komunikuje s dotykovou obrazovkou pomocou *S7* protokolu, keďže sa jedná o zariadenia od spoločnosti *Siemens*.

1.4 Popis jednotlivých procesných buniek

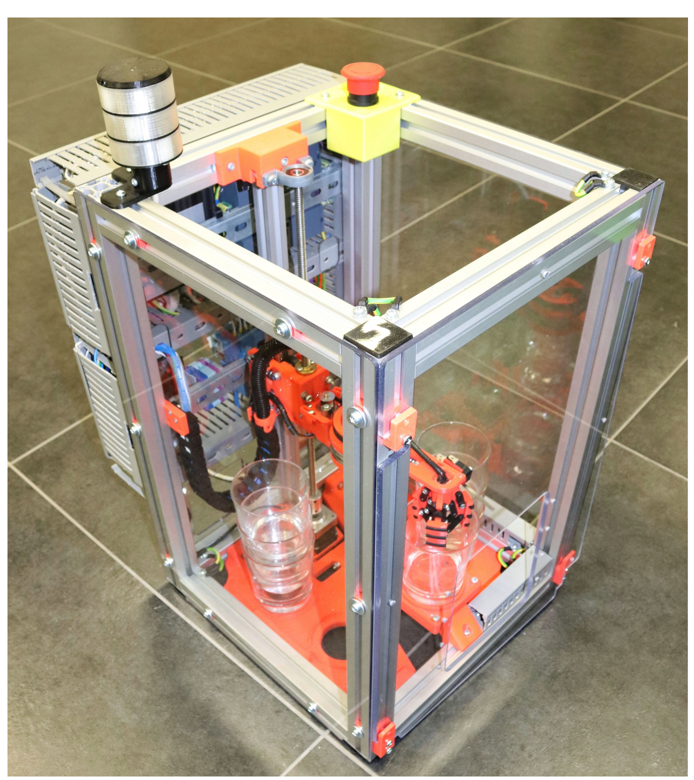
Barman disponuje nasledovnými bunkami:

- Sklad pohárov
- Sklad alkoholu
- Drvič ľadu
- Sódovač
- Shaker
- Transport - robotické rameno *SCARA*

1.4.1 Sklad pohárov

Sklad pohárov má za úlohu vydávať poháre a nahrávať receptúru do *RFID* čipu na základe objednávok z nadradeného systému. Taktiež prijíma použité poháre. Nachádza sa na stole a pohár stráca kontakt s čítačkou pri manipulácii v rámci bunky.

Vo všeobecnosti ponúka službu: `Výdaj`, `Príjem`. Dokáže manipulovať s jedným pohárom naraz.



Obr. 1.3: Procesná bunka - `Sklad pohárov` [6].

1.4.2 Sklad alkoholu

Táto bunka má za úlohu skladovať a dávkovať alkoholické nápoje a sirupy. Je najväčšia zo všetkých a je umiestnená zo zadnej časti stolu. Pohár stráca kontakt s čítačkou pri manipulácii v rámci bunky.

Ponúka akciu `Nalej` a ako parameter tejto funkcie sú informácie o množstve a produkte, ktoré má byť naliate.

Obr. 1.4: Procesná bunka - `Sklad alkoholu` [7].

1.4.3 Drvič ľadu

Táto bunka skladuje kocky ľadu, ktoré následne dávkuje vo forme ľadovej triešti. V tejto bunke pohár nestráca kontakt s čítačkou.

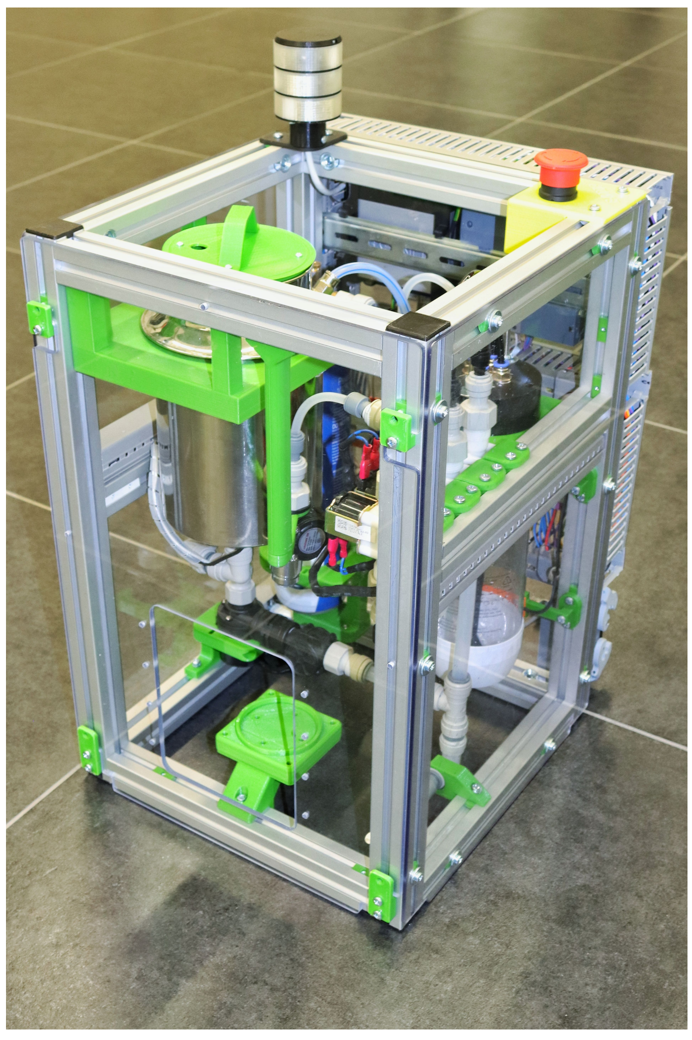
Ponúka akciu `Pridaj ľad` a ako parameter tejto funkcie sa udáva množstvo ľadu v gramoch.

Obr. 1.5: Procesná bunka - `Drvič ľadu` [8].

1.4.4 Sódovač

Táto bunka vyrába sódu, pomocou stlačeného CO_2 v tlakovej nádobe. V tejto bunke pohár nestráca kontakt s čítačkou.

Ponúka akciu `Pridaj sódu` a ako parameter tejto funkcie sa udáva množstvo sódy v decilitroch.



Obr. 1.6: Procesná bunka - `Sódovač` [9].

1.4.5 Shaker

Shaker slúži na premiešanie nápoju v pohári. V tejto bunke stratí čítačka kontakt s *RFID* čipom umiestneným na spodnej strane pohára.

Ponúka akciu `Miešaj` a ako parameter tejto funkcie je čas miešania.

Obr. 1.7: Procesná bunka - `Shaker` [10].

1.4.6 Robotické rameno *SCARA*

Táto bunka sa skladá z dvoch častí a to z robotického ramena *SCARA*, ktoré slúži na presúvanie produktov medzi bunkami a z dopravníka, ktorý slúži na výdaj vyrobených nápojov a príjem prázdnych pohárov.

Robot ponúka akciu `Transport` a ako vstupné parametre má bod, z ktorého má vyzdvihnúť pohár a bod, kde ho má položiť. V rámci *test bedu* sú už pozície pre pohár presne určené a robot ich má pevne označené a naučené. Keďže majú bunky umiestnené na stole otvor pre pohár na rovnakom mieste, vieme ich zamieňať medzi sebou.

Obr. 1.8: Transportná bunka - `Robotické rameno` ***SCARA*** [11].

1.5 Proces výroby nápoja

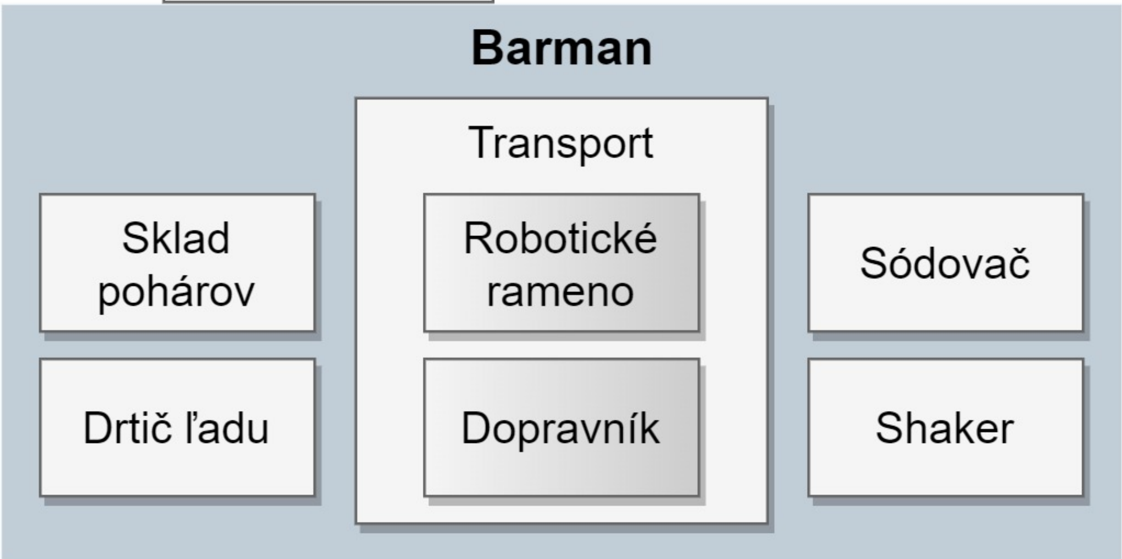
Celá výroba začína v sklade pohárov, kde z nadradeného systému príde objednávka a nahrá do *RFID* čipu objednanú receptúru. V tejto pamäti budú uložené dáta nielen o receptúre, ale aj o stave v akom sa výrobok nachádza. Tým pádom celá výroba nebude závisieť na centralizovanom systéme, kde bude uložený stav každého výrobku, ale každý výrobok bude mať uložené údaje o jeho stave v sebe. Komunikačný modul výrobku bude aktívny len v prípade, keď sa pohár bude nachádzať na *RFID* čítačke. Vtedy sa vyčítajú dáta a komunikačný modul výrobku na základe týchto dát bude riadiť výrobu ďalej. Ďalší požiadavok je, aby modul produktu vedel v rámci lokálnej siete vyhľadať dostupné bunky podľa nejakých kategórii a následne s nimi komunikoval. Najlepšie by bolo, aby tento zoznam buniek nebol centralizovaný, teda aby výroba nezáležala na správnej funkcii jedného systému. Po nájdení správnej bunky si výrobok zistí, či mu je schopná bunka splniť požiadavok. Ak áno, tak si rezervuje u nej službu a čaká na to, kým príde na rad. Keď príde na rad, dohodne si transport a po príchode na správne miesto si výrobok spustí danú službu. Po dokončení sa tieto kroky opakujú až pokiaľ nebude výrobok hotový. Výrobok bude predstavovať klientskú aplikáciu, ktorá bude komunikovať s bunkami vo výrobe (servery). Nadradený systém vie meniť poradie objednávok tak, aby sa vykonávali čo najefektívnejšie. Ďalej by sa mal tento pohár vedieť pohybovať medzi výrobnými bunkami, ktoré budú spracovávať jeho požiadavky a nakoniec sa pohár predá zákazníkovi pomocou robotického ramena na dopravník. Na nasledujúcom obrázku môžeme vidieť blokovú schému rozloženia buniek v *test bede* [1].

V rámci *test bedu* sa počíta s náhodným množstvom a príchodom objednávok. Viacero produktov môže byť obsluhovaných naraz a v rôznom poradí. Niektoré receptúry budú mať určené presné poradie úkonov, a niektoré úkony sa budú môcť vykonávať v ľubovolnom poradí. Keďže ide len o *test bed*, každý typ bunky je vytvorený len raz, ale v reálnej továrni by bolo viacej buniek, ktoré by vykonávali tu istú činnosť. Tak isto by tu mohlo byť viac typov buniek, ktoré ponúkajú iné služby.

1.6 Všeobecné zhrnutie

Všetky procesné bunky vlastne ponúkajú nejakú službu, alebo viac služieb, a tieto služby sú parametrizovateľné. Na základe tohto zistenia môžem určiť jednotné rozhranie pre spúšťanie služby pohárom.

Keďže bunky majú napísaný program podľa štandardu *S88*, jednotlivé služby sú naprogramované ako fázy, ktoré ďalej ovládajú riadiace moduly a tie sú prepojené s fyzickými vstupmi a výstupmi. Tieto fáze majú automatické a manuálne riadenie. V automatickom riadení riadi fázu nadradený systém. V našom prípade to je produkt.



Obr. 1.9: Bloková schéma rozloženia procesných buniek.

V ďalších kapitolách bude rozobrané ako by takáto komunikácia mohla prebiehať.

Ďalej si môžeme bunky rozdeliť do skupín podľa typu služby, ktoré ponúkajú:

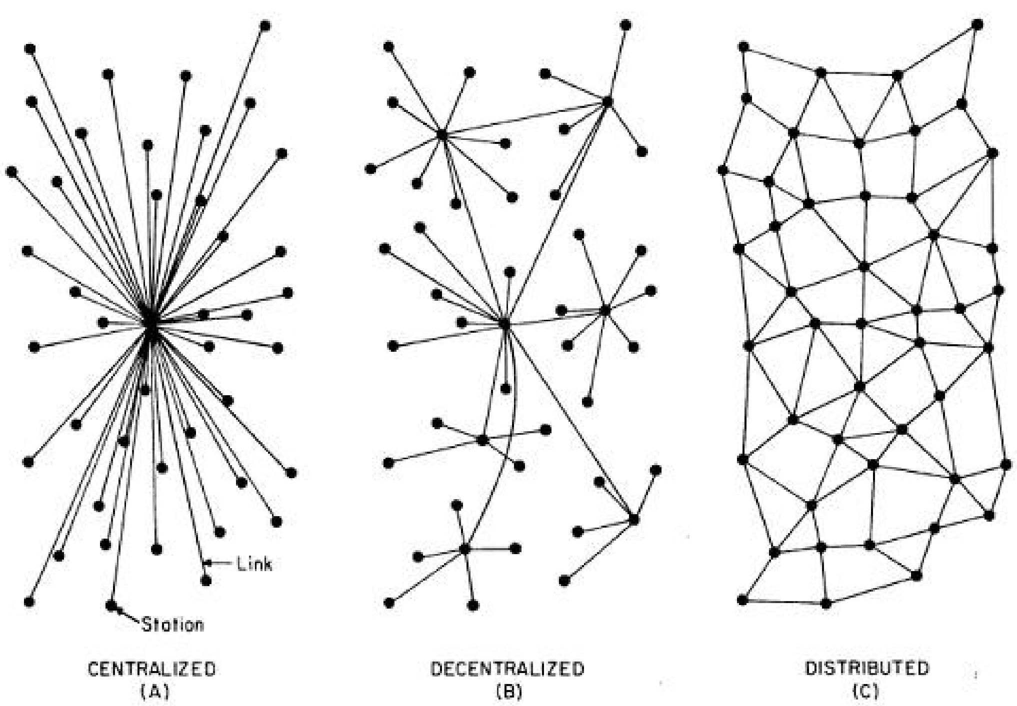
- Transportné - vyznačujú sa tým, že presúvajú produkty
- Skladovacie - vyznačujú sa poskytnutím materiálu
- Výrobné - vyznačujú sa aplikovaním nejakého výrobného procesu

1.7 Požiadavky na distribuované riadenie

Na nasledujúcom obrázku môžeme vidieť rozdiel medzi centralizovaným, decentralizovaným a distribuovaným riadením. Hlavným rozdielom je, ako a kde sa vykonávajú rozhodnutia, a ako sa zdieľajú informácie medzi riadiacimi systémami. V centralizovanom riadení sú všetky prvky výroby ovládané z jedného miesta a pri decentralizovanom riadení sa hlavné riadenie rozdelí na menšie časti, čo sú vlastne menšie centralizované riadenia prepojené medzi sebou. V distribuovanom riadení vie komunikovať každý element výroby s každým. Rozhodovanie je posunuté od riadenia jedným nadriadeným systémom smerom ku komponentom produkčného systému ako sú stroje, produkty a iné, čiže môžu robiť vlastné rozhodnutia bez nadradeného systému. Rozhodovanie a ovládanie procesu je vykonávané autonómne a paralelne,

v každom jednom subsystéme, ktoré spolu navzájom komunikujú cez sieť. Ako bolo spomenuté pri *Industry 4.0*, jednotlivé komponenty potrebujú mať možnosť spracovávať informácie a vykonávať operácie na základe vlastného rozhodnutia, čiže musia byť autonómne. A samozrejme, všetky komponenty musia so sebou vedieť komunikovať a vymieňať si medzi sebou informácie, na základe ktorých sa budú rozhodovať. Distribuované riadenie je dobré pre dynamicky meniace sa prostredia, kde treba na tieto zmeny reagovať rýchlo. Okrem toho tiež distribuovaná štruktúra propaguje ďalšie prospešné vlastnosti a ponúka možnosť seba konfigurácie, pohotovosti a adaptácie v reálnom čase na neočakávané udalosti, ako sú návaly objednávok alebo výpadok stroja.

Hlavnou nevýhodou decentralizovaných systémov je zložitá koordinácia jednotlivých komponentov vo výrobe, lebo každý sa snaží dosiahnuť svoj cieľ [2] [3].



Obr. 1.10: Rozdiel medzi typmi riadenia výroby [3].

2 Výber komunikačného protokolu

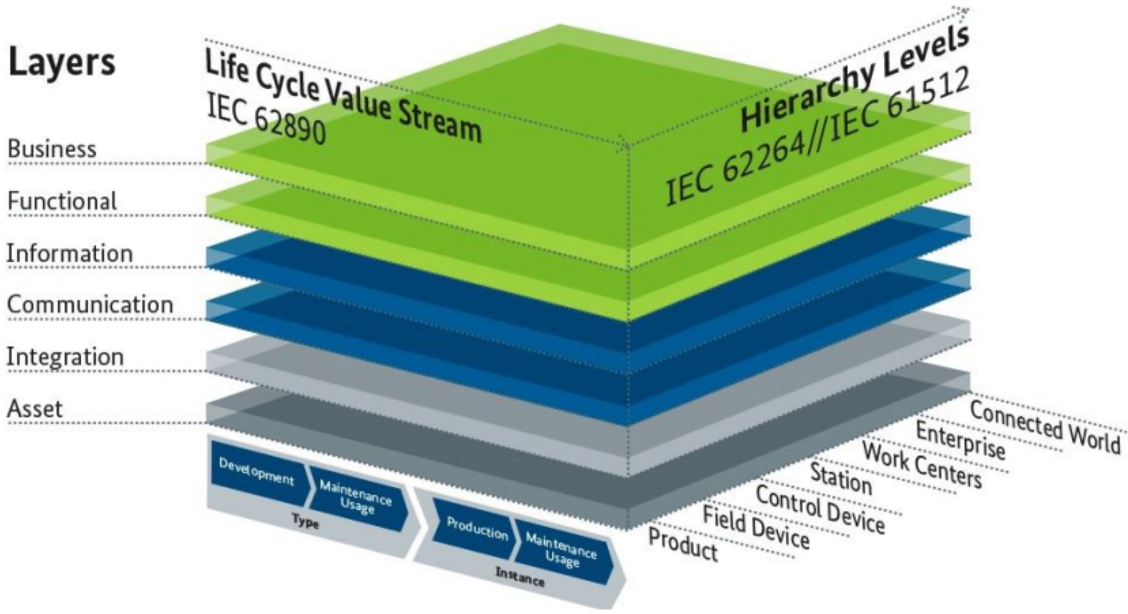
Vybraný komunikačný protokol bude slúžiť na komunikáciu medzi procesnými bunkami vo výrobe a produktami. Pri výbere budem dbať nato, aby bol tento protokol otvorený s dobrou dokumentáciou a hlavne, aby splňoval požiadavky konceptu *Industry 4.0*.

V našom prípade disponujú bunky riadiacim systémom *PLC S7-1200*, ktorý má k dispozícii nasledujúce komunikačné protokoly:

- `S7 protokol` - Komunikačný protokol vyvinutý spoločnosťou *Siemens*, ktorý slúži primárne na komunikáciu medzi *Siemens* zariadeniami. Je to zatvorený protokol, čo znamená, že k nemu nie je dostupná dokumentácia.
- `TCP/UDP` - Posielanie dát pomocou *TCP/UDP* transportných protokolov na konkrétnu *IP* adresu a port.
- `Modbus TCP` - Aplikačný protokol, ktorý prenáša dáta pomocou *TCP/IP* protokolu.
- `Modbus RTU` - Taktiež aplikačný protokol, ale pracuje na sériovom rozhraní s inou fyzickou vrstvou a má odlišný rámec než *Modbus TCP*.
- `PROFINET` - Nástupca `PROFIBUS`. Je kompatibilný s klasickým *TCP/IP* protokolom a slúži hlavne na prenášanie dát v reálnom čase (riadenie pohonov).

Uvedené komunikačné protokoly nespĺňajú požiadavky *Industry 4.0*. Dôvod je, že bežné priemyselné komunikačné protokoly umožňujú posielať dáta, napríklad teplota vzduchu bez ďalších meta dát, ako napríklad fyzikálna jednotka, v akej sa teplota meria, rozsah snímaču a iné. Medzi hlavné požiadavky komunikačného protokolu patrí vysoká bezpečnosť a škálovateľnosť. To znamená, aby bol komplexný na toľko, aby sa používal v podnikových aplikáciach ako aj v jednoduchých embedded zariadeniach. Bezproblémová výmena informácií medzi fyzickými subjektmi v továrni a IT systémami je nevyhnutná na umožnenie kyber-fyzikálnych produkčných systémov vo výrobe.

V roku 2016 zaviedli skupiny, ktoré vyvíjajú *Industry 4.0* nový pojem *AAS - Asset administration shell*, čo v preklade znamená digitálna obálka aktíva, v rámci ktorého bol navrhnutý model referenčnej architektúry *RAMI 4.0*.



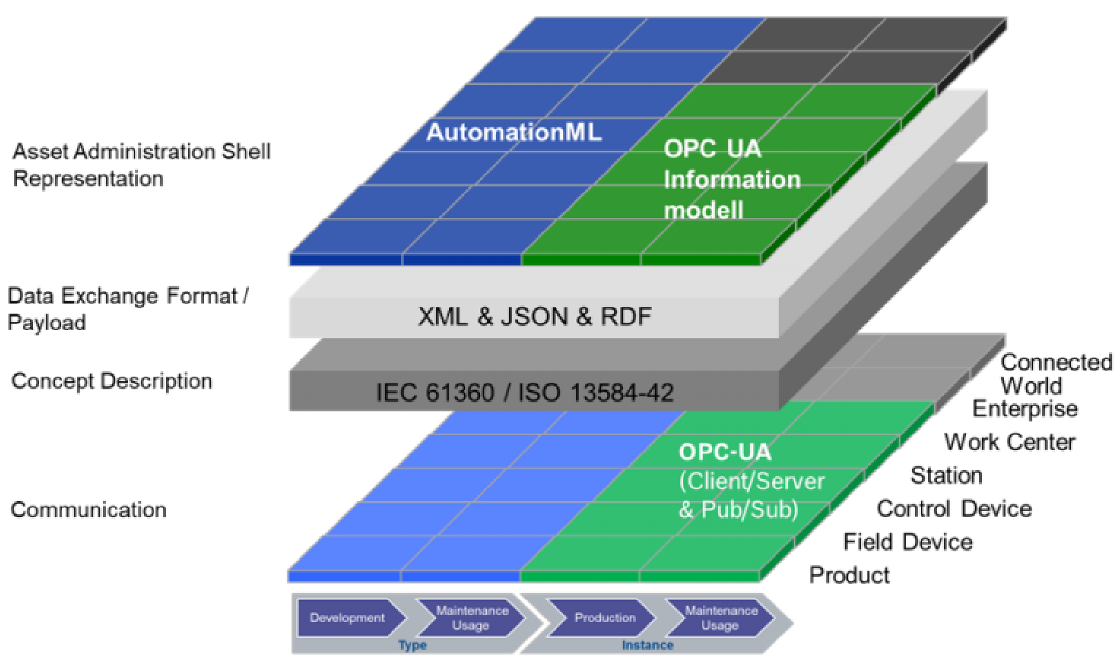
Obr. 2.1: Referenčný 3D model *RAMI 4.0* [5].

V podstate to znamená, že každé aktívum môže mať svoj digitálny obraz, či to je nejaký softvér, hardvér alebo celý stroj. V rámci tejto práce štandardizovali ako jeden z komunikačných protokolov práve *OPC UA*. Tento protokol používa architektúru klient-server. Toto využitie je vidieť na nasledujúcom obrázku 2.2.

Práve z týchto dôvodov som zvolil ako komunikačný protokol *OPC UA*. V rámci *test bedu* budú bunky predstavovať server, to znamená, že budú sprístupňovať dáta a služby ostatným zariadeniam vo výrobe. Keďže riadiaci systém v bunke nedokáže komunikovať pomocou *OPC UA*, tak komunikačný modul bunky sa nemôže nachádzať v *PLC*. Preto budem musieť vytvoriť aplikáciu, ktorá bude pomocou štandardného priemyselného komunikačného protokolu získavať dáta z *PLC*, ktoré sa budú ďalej používať v *OPC UA* komunikácii. Na základe vyššie spomenutých protokolov, ktoré sa nachádzajú v riadiacom systéme, som zvolil *Modbus TCP*. Tento protokol je otvorený, platformovo nezávislý a veľmi často používaný v štandardných riadiacich systémoch na trhu. To, aké dáta sa budú komunikovať, určím v ďalších kapitolách, kde budem vytvárať štandardnú dátovú štruktúru bunky pre komunikačný modul. Najdôležitejšou úlohou tejto aplikácie je vytvoriť bránu (`gateway`), ktorá by prepojila rôzne zariadenia z prostredia *Industry 4.0*.

Komunikačný modul produktu bude predstavovať *OPC UA* klienta, to znamená, že bude iniciovať komunikáciu a riadiť výrobu nápoja podľa receptúry. Ďalej tento modul bude dostávať dáta z *RFID* čítačky pomocou *MQTT* protokolu.

Tieto aplikácie budú bežať na *Nano Pi*. Na obrázku 2.3 sa nachádza bloková schéma zariadení, spolu s vyznačenými aplikáciami a komunikáciami medzi nimi.



Obr. 2.2: Využitie *OPC UA* v *RAMI 4.0* v komunikačnej a reprezentačnej vrstve [4].

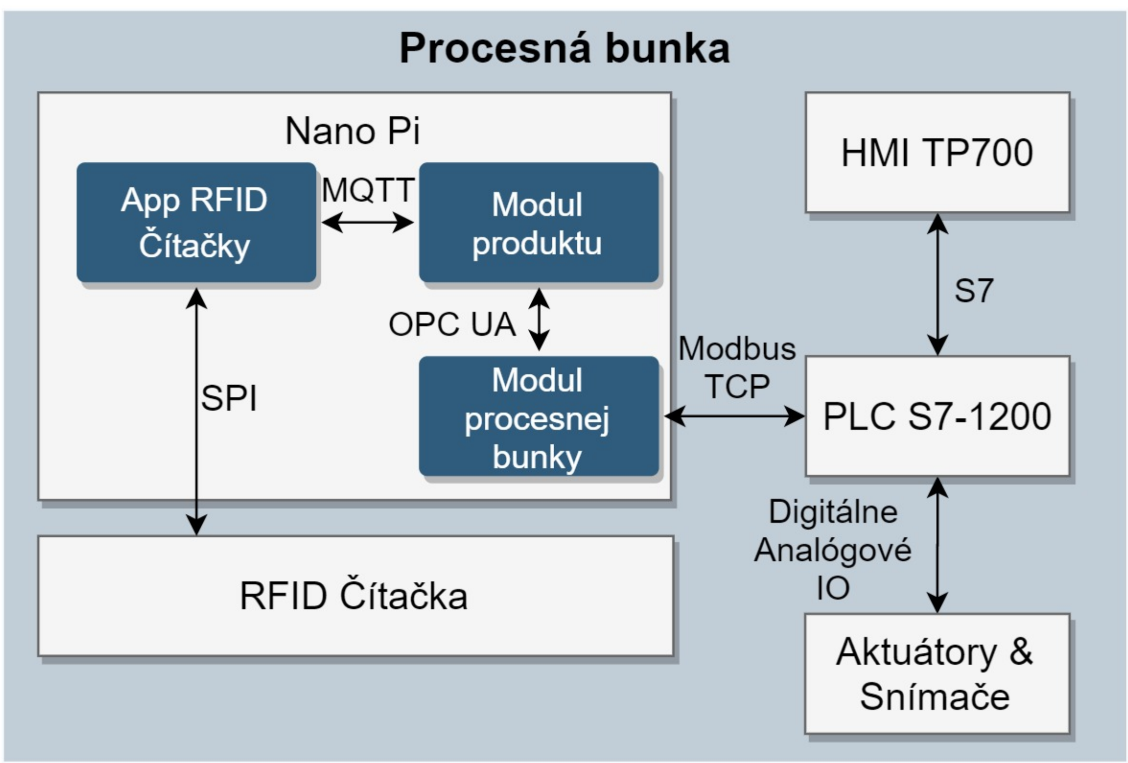
2.1 Výber implementácie *OPC UA*

Predtým, než začnem vytvárať aplikácie, musím nájsť vhodnú implementáciu *OPC UA* protokolu, ktorú použijem v riešení. Tento komunikačný štandard je implementovaný skoro v každom známom programovacom jazyku. Zoznam *OpenSource* implementácii môžete nájsť na oficiálnej stránke *OPC Foundation*.

Pri výbere som sa držal toho, aby bola knižnica *OpenSource*, aktívne vyvíjaná a aby bola napísaná v programovacom jazyku, ktorý je platformovo nezávislý. Zároveň musí byť táto knižnica spustiteľná na embedded zariadeniach. Ďalej som dbal nato, aby v nej bolo implementovaných čo najviac definovaných servisov podľa *OPC UA* štandardu. Potreboval som hlavne servisy, ktoré vyplynuli zo zadania v sekcii 1.5, kde bol popísaný postup výroby nápoja. Tieto servisy sú nasledovné:

- Vytvorenie dátovej štruktúry bunky - `Node Management Service Set`
- Notifikácie o zmenách - `Subscription Service Set`
- Vyhľadanie dostupných buniek v sieti, podľa kategórii - `Discovery Service Set`, kde je implementovaný `Local Discovery Server - Multicast Extension` (*LDS-ME*)

Nakoľko bol *LDS-ME* doplnený do štandardu len v posledných rokoch, väčšina knižníc túto funkciu neobsahuje. Výnimkou sú *Open62541*, implementovaný v *C99/C++98* a *Eclipse Milo*, implementovaný v *Java 8*. Tento servis pridal do knižníc ten istý člo-



Obr. 2.3: Bloková schéma zariadení a komunikácií medzi aplikáciami v rámci jednej bunky.

vek, *Stefan Profanter*, ktorý implementoval tieto funkcie v rámci článku [15]. Po naštudovaní dokumentácii k jednotlivým knižniciam, vytvorení jednoduchých aplikácii a prezretí git repozitáru som zistil, že implementácia *LDS-ME* sa v knižnici *Eclipse Milo* nedostala do hlavnej verzie a taktiež je pomalšie vyvíjaná a má trochu inú architektúru než *Open62541*.
Z týchto dôvodov som zvolil knižnicu *Open62541*. Má nasledovné vlastnosti:

- `Škálovateľnosť` - Tvorcovia knižnice zvolili architektúru aplikácie tak, aby splňovala špecifikáciu a to tým, že si vieme určiť aké servisy chceme používať v našej aplikácii a ostatné sa nevygenerujú do zdrojového kódu. Tento proces je vysvetlený v nasledujúcej kapitole.
- `C99/C++98` - Knižnica je naschvál napísaná v starších špecifikáciách *C/C++*, aby ju bolo možné spustiť aj na starších embedded zariadeniach, ktoré disponujú staršími prekladačmi.
- `Udalosťami riadená architektúra`
- `OpenSource` - Používanie knižnice v aplikácii je bezplatné, zdrojové kódy knižnice sú voľne dostupné a je možné sa podieľať na vývoji.
- `Editovateľný informačný model` - Informačný model je možné editovať aj

počas toho ako je server spustený a to nielen zo servera ale aj z klienta.

- Aktuálna verzia `1.0.1` dostala oficiálnu certifikáciu od *OPC Foundation*.

3 Návrh informačného modelu bunky

Jednou silnou stránkou *OPC UA* je sémantický adresový priestor. Informačný model definuje uzly a ich štruktúru definovanú v adresovom priestore. Je to podobné ako pri OOP, informačný model definuje typy, ktoré môžu byť rozšíriteľné (dedičnosť) a môžu byť vytvorené objekty týchto typov. Tak isto ponúka možnosť vytvárať metódy, s voliteľnými vstupmi a výstupmi. *OPC UA Foundation* definuje základný informačný model, ktorý obsahuje základné dátové typy, referencie, objekty a iné prvky. Tento základný informačný model ďalej rozširujú takzvané *companion specification*, čo v preklade znamená špecifikácie spoločníkov. V podstate rozširujú základné typy o typy vlastných objektov alebo definuje nové typy premenných. List oficiálne podporovaných špecifikácii nájdete na nasledujúcej adrese https://opcfoundation.org/developer-tools/specifications-opc-ua-information-models. Tieto špecifikácie vznikajú preto, aby sa dáta modelovali rovnakým spôsobom medzi rôznymi výrobcami priemyselných zariadení.[14].

Adresový priestor je zložený z takzvaných uzlov (*Nodes*), ktoré majú medzi sebou referencie (určujú vzťah medzi uzlami) a dokopy tvoria stromovú štruktúru.

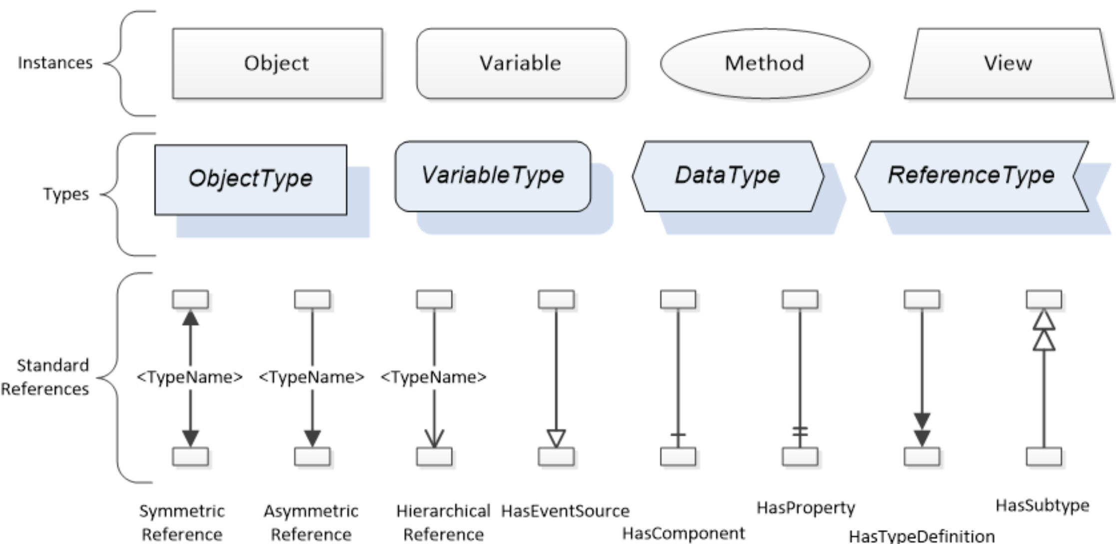
OPC UA ďalej definuje niekoľko generických služieb, na získavanie dát, úpravu a prehľadávanie adresového priestoru. Generických preto, že v adresovom priestore môžeme mať vytvorené vlastné dátové typy, objekty, premenné a klient si môže tieto dáta vždy vyčítať bez predošlej znalosti týchto informácií a pomocou jednej funkcie so správnymi parametrami.

V mojom prípade vytvorím informačný model, ktorý bude definovať všeobecne procesnú bunku a budem s týmto modelom môcť popísať všetky bunky, ktoré sa nachádzajú v *Barmanovi*. Táto definícia bude obsahovať nové typy objektov, ktoré budú vyjadrovať potrebné údaje (premenné a vlastnosti) a služby (metódy), ktoré bunka ponúka.

Na nasledujúcom obrázku 3.1 môžete vidieť zhrnutie grafických symbolov, ktoré sa využívajú pri grafickom znázornení informačných modelov.

3.1 Všeobecný postup pri vytváraní informačných modelov

Vlastný informačný model, ktorý obsahuje nové dátové typy, typy objektov, referencie a iné prvky sa dá vytvoriť dvoma spôsobmi, závisí to od toho, akú *OPC UA* knižnicu si vyberieme pre programovanie aplikácie. V našom prípade používame knižnicu



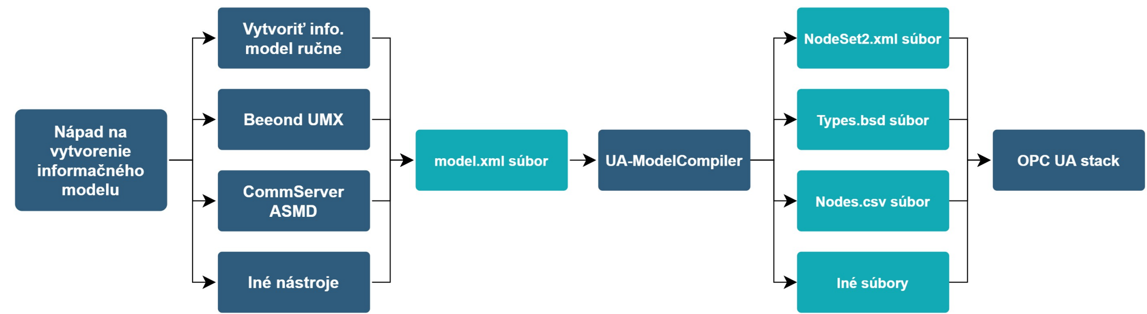
Obr. 3.1: Grafické symboly [12].

Open62541, ktorá podporuje oba spôsoby. Má *API* na vytváranie informačných modelov pomocou kódu a druhý spôsob je definovať `nameModel.xml` súbor, v ktorom sa bude nachádzať definícia nášho informačného modelu. Celý návrh vlastného informačného modelu začína pri myšlienke toho, čo chcem popísať. Túto myšlienku nám predstavuje `model.xml` súbor. Tento *xml* súbor môžeme vytvoriť buď ručne alebo pomocou takzvaných *OPC UA* modelerov, v ktorých sa definuje štruktúra dátových typov a inštancii v grafickom prostredí a následne sa vygeneruje niekoľko potrebných súborov s aplikáciou *UA-ModelCompiler*, ktorá vytvorí štandardnú štruktúru adresového priestoru. Tento prekladač vygeneruje niekoľko súborov. Najdôležitejší z nich má názov `name.NodeSet2.xml`. Tento súbor použijeme na vygenerovanie *C* kódu, ktorý použijeme v našej aplikácii. Na vygenerovanie *C* kódu slúži *Open62541 Nodeset Compiler*. Nevýhodou riešenia s použitím server *API* je, že informačný model nie je prenosný. Na nasledujúcom obrázku môžeme vidieť zhrnutý pomyselný postup pri vytváraní vlastného informačného modelu [13].

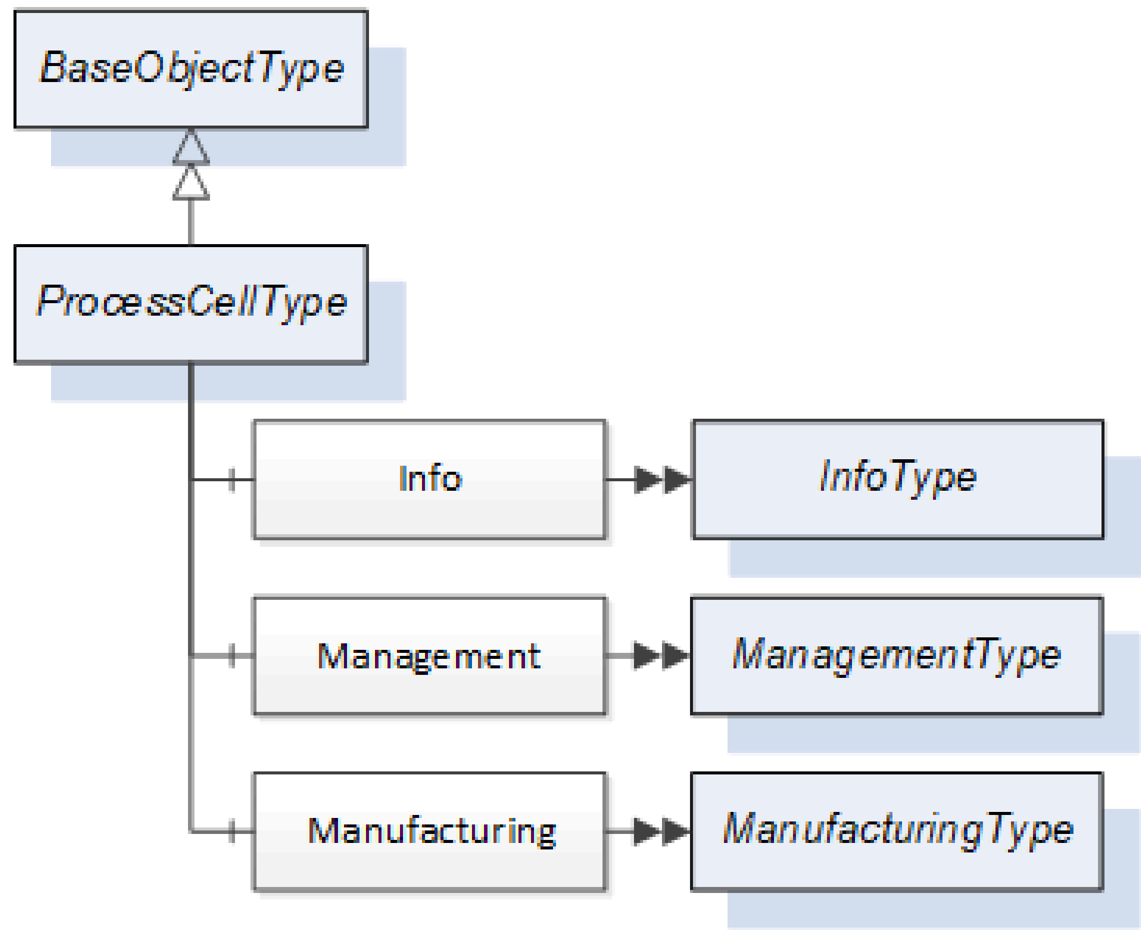
3.2 Informačný model procesnej bunky

V tejto sekcii bude popísaný tvar informačného modelu bunky a vysvetlené jednotlivé funkcie a významy uzlov.

Celý model bude zastrešovať objekt `ProcessCellType`, ktorý obsahuje ďalšie objekty ako `Info`, kde sa nachádzajú všeobecné informácie, `Management` má na starosti rezervácie, ktoré si vytvára sám produkt a `Manufacturing` má za úlohu ovládať výrobu a poskytovať informácie o výrobe.

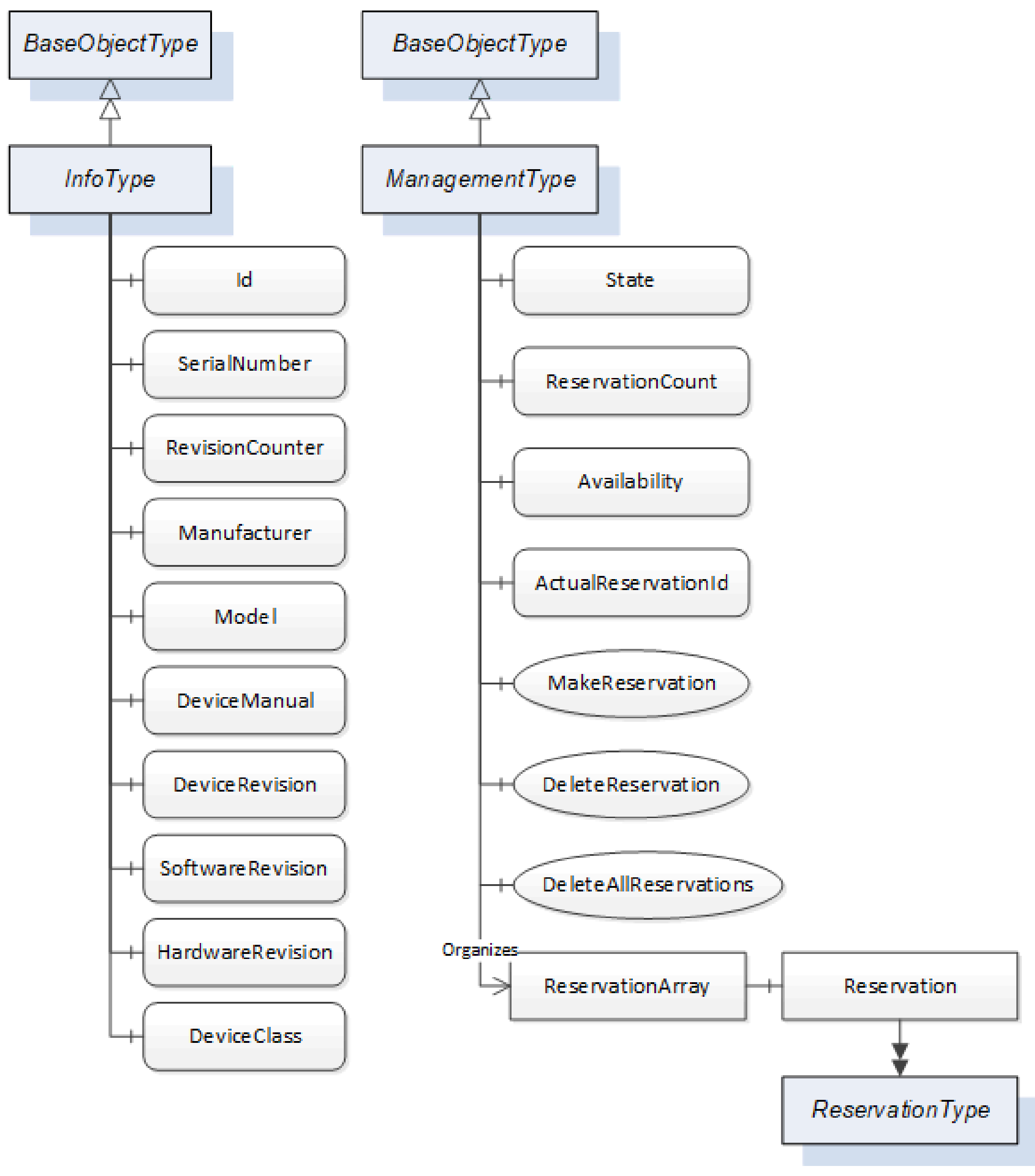


Obr. 3.2: Postup pri vytváraní informačného modelu.



Obr. 3.3: Informačný model procesnej bunky.

Na nasledujúcom obrázku 3.4 môžeme vidieť detailný informačný model objektu `Info` a `Management`. Objekty sa skladajú z premenných, ktoré sú pripojené referenciou `HasComponent`. Výnimkou je pri objekte `ManagementType`, kde má objekt `ReservationArray` referenciu `Organizes` a obsahuje všetky vytvorené rezervácie. Dátový typ rezervácie (`Reservation`) je ukázaný na nasledujúcom obrázku 3.5.



Obr. 3.4: Informačný model dátového typu info a manažment.

Ďalej sa v objekte `ManagementType` nachádza celkový stav a dostupnosť tejto služby s dátovým typom `int32`. Taktiež tu je celkový počet rezervácii s dátovým typom `int32` a jedinečný identifikátor aktuálnej rezervácie, ktorá je na rade s dátovým typom `uint64`. A nakoniec sa tu nachádzajú tri metódy:

- `MakeReservation` - Táto metóda slúži na vytvorenie novej rezervácie a jej vstupné parametre sú:
  - `Product Id` - Jedinečný identifikátor produktu, ktorý si rezervoval danú

akciu.

  - `Action Id` - Id rezervovanej akcie.
  - `Operation Order` - Poradie danej operácie v receptúre.
  - `Parameter A` - Vstupný parameter `A` pre akciu.
  - `Parameter B` - Vstupný parameter `B` pre akciu.
- `DeleteReservation` - Táto metóda má ako vstupný parameter jedinečný identifikátor rezervácie, ktorý sa má zmazať a vracia len boolovskú hodnotu o tom či bolo mazanie úspešne.
- `DeleteReservations` - Táto metóda nemá žiadne vstupné parametre a vracia len boolovskú hodnotu o tom či sa zmazali všetky rezervácie úspešne.

V objekte `Info` sa nachádzajú všeobecné informácie o bunke podľa oficiálnej špecifikácie modelovania zariadení [17].

- `Id` - Jedinečný identifikátor produktu. Dátový typ `uint64`.
- `SerialNumber` - Jedinečný výrobný identifikátor produktu. Jedná sa o kód, ktorý sa nachádza na obale produktu. Dátový typ `string`.
- `RevisionCounter` - Počítadlo počítajúce počet zmien konfiguračných dát. Napríklad, koľkokrát sa prestavila fyzikálna jednotka meranej teploty na senzore teploty. Ak nie je počítadlo implementované, východzia hodnota je *-1*. Dátový typ `int32`.
- `Manufacturer` - Názov spoločnosti, ktorá vyrobila dané zariadenie. Dátový typ `string`.
- `Model` - Názov produktu. Dátový typ `string`.
- `DeviceManual` - *Url* adresa, alebo cesta k súboru s manuálom. Dátový typ `string`.
- `DeviceRevision` - Celková revízia zariadenia (firmwaru). Dátový typ `string`.
- `SoftwareRevision` - Verzia alebo revízia softwaru (firmwaru). Dátový typ `string`.
- `HardwareRevision` - Revízia hardwaru. Dátový typ *string*.
- `DeviceClass` - Udáva z akej oblasti je dané zariadenie. Dátový typ `string`.

Na nasledujúcom obrázku 3.5 môžeme vidieť detailný informačný model objektu `Manufacturing` a `Reservation`. Oba objekty majú len premenné s referenciou *HasComponent*.

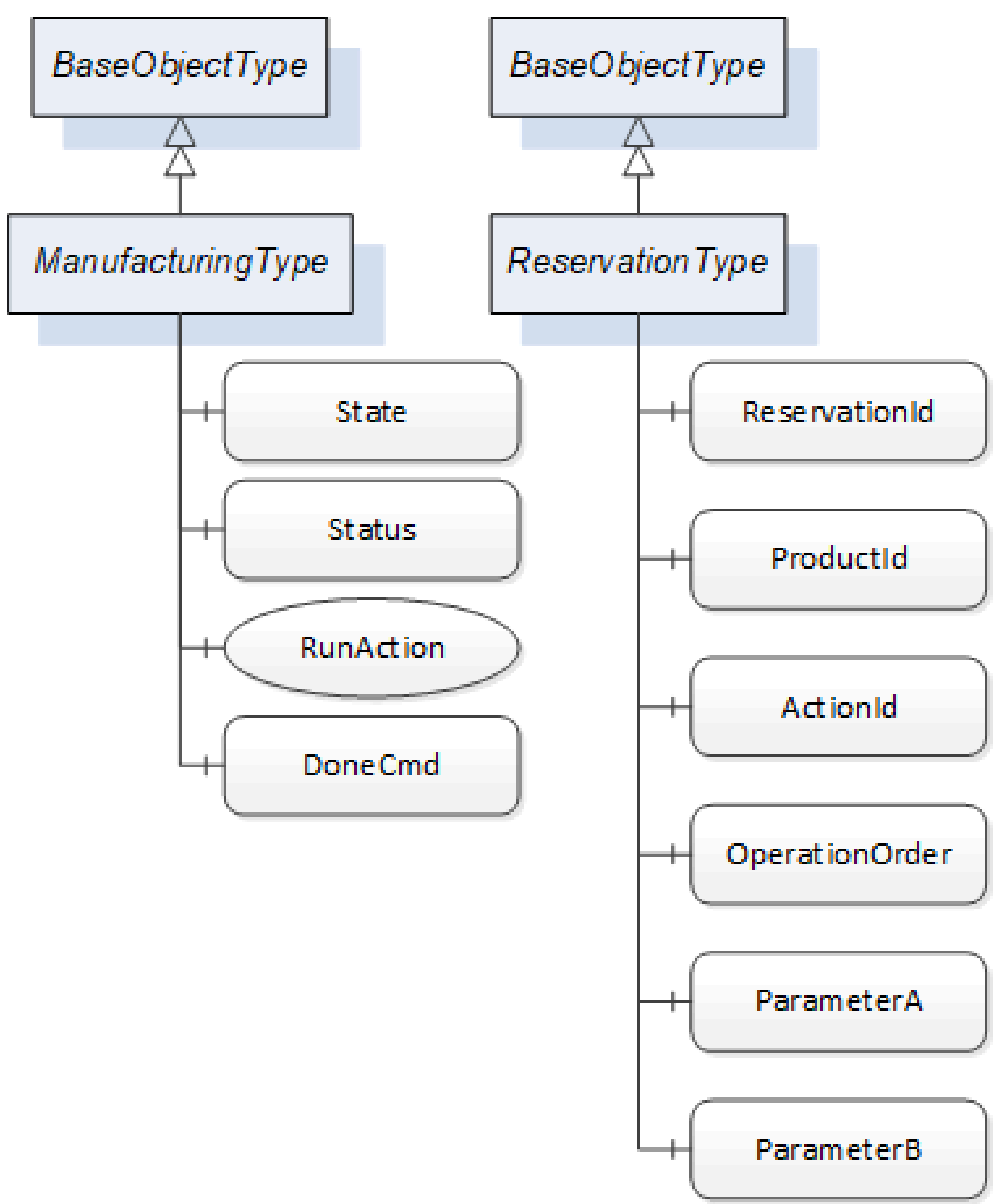
Objekt `Manufacturing` obsahuje premenné:

- `State` - Stav výroby s dátovým typom `uint16`. Nadobúda stavy:
  - `0 : Waiting` - Čaká na spustenie akcie.
  - `10 : Working` - Práve vykonáva akciu.
  - `20 : Done` - Hotovo, premenná *Status* obsahuje výsledok akcie.

- `Status` - Status výroby s dátovým typom `uint16`. Zatiaľ sú definované len všeobecné výsledky a je možnosť doplniť pre každú operáciu špecifickejšie výsledky.
  - `0 : None` - Predstavuje *null*, nevykonalo sa.
  - `1 : OK` - Akcia dopadla v poriadku.
  - `5 : NOK` - Daná akcia zlyhala.
- `RunAction` - Metóda s ktorou sa spúšťa výroba. Ako vstupné parametre má:
  - `Action Id` : Id akcie, ktorá sa má vykonať.
  - `Parameter A` : Vstupný parameter `A` pre akciu.
  - `Parameter B` : Vstupný parameter `B` pre akciu.
- `DoneCmd` - Príkaz na ukončenie výroby, keď produkt spracoval výsledok akcie a odchádza zo vstupu procesnej bunky. Po príchode tohto príkazu sa zmení stav výroby z `Done` na `Waiting`.

Po vytvorení rezervácie sa vytvorí objekt s jedinečným `ReservationId` a ostatné parametre sa skopírujú zo vstupných parametrov metódy `MakeReservation`.

Tvorenie informačného modelu pomocou knižnice *Open62541* bude popísané v kapitole 5.3.



Obr. 3.5: Informačný model dátového typu výroby a rezervácie.

4 Receptúra

Pred začatím programovania komunikačného modulu výrobku som musel najprv definovať tvar receptúry a stavových dat, ktoré budú uložené v *RFID* čipe.

V tomto riešení sa používajú *RFID* čipy of spoločnosti *NXP Semiconductors* s názvom *NTAG213/215/216*. Tieto čipy majú pamäť organizovanú po štvor bytových stránkach. Spomenuté typy čipov sa líšia veľkosťou užívateľskej pamäte a to *NTAG213* má k dispozícii 45, *NTAG215* má 135 a *NTAG216* 231 stránok. Štruktúra pamäte vyzerá nasledovne. Je identická aj pre ďalšie spomenuté čipy, až na veľkosť užívateľskej pamäte [19].

Page Adr		Byte number within a page				
Dec	Hex	0	1	2	3	Description
0	0h	serial number				Manufacturer data and static lock bytes
1	1h	serial number				
2	2h	serial number	internal	lock bytes	lock bytes	
3	3h	Capability Container (CC)				Capability Container
4	4h	user memory				User memory pages
5	5h					
...	...					
38	26h					
39	27h					
40	28h	dynamic lock bytes			RFUI	Dynamic lock bytes
41	29h	CFG 0				Configuration pages
42	2Ah	CFG 1				
43	2Bh	PWD				
44	2Ch	PACK		RFUI		

Obr. 4.1: Štruktúra pamäte čipu *NTAG213* [19].

Každý čip má svoje jedinečné sedem bytové sériové číslo, ktoré sa nachádza v oblasti statických dát, ktoré sú chránené proti zápisu. Na obrázku je naznačených deväť bytov, ale posledné dva byty sú kontrolné.

Tvorba *RFID* čítačky, výberu čipov a aplikácie pre obsluhu čítačky nie je súčasťou mojej práce, ale vzhľadom k tomu, že to úzko súvisí s mojou prácou a v daných čipoch sa budú ukladať stavové dáta a receptúra k danému produktu, podieľal som sa na tvorbe *API*, ktoré slúži na získavanie a zápis dát do *RFID* čipu. Výmena dát prebieha pomocou *MQTT* protokolu ako už bolo znázornené na obrázku 2.3.

4.1 *API* obslužnej aplikácie *RFID* čítačky

Ako bolo spomínané, komunikačné moduly produktu a procesnej bunky spolu s obslužnou aplikáciou *RFID* čítačky budú spustené na *Nano Pi*. Spolu s týmito apli-

káciami tam bude spustený aj *MQTT* prostredník (*broker*), pomocou ktorého bude prebiehať výmena dát z *RFID* čítačky. Obslužná aplikácia bude na dopredu definované koncové body, alebo tiež *"témy"*, posielať dáta na prostredníka a komunikačný modul výrobku bude zaregistrovaný na týchto koncových bodoch a bude dostávať dáta. Úlohou prostredníka je vlastne prijímať dáta a následne ich rozosielať zaregistrovaným účastníkom.

Definované sú nasledovné *MQTT* koncové body, pomocou ktorých sa budú prijímať alebo zapisovať dáta.

Koncové body, na ktoré posiela obslužná aplikácia čítačky dáta:

- `spi/msg` - Logovacie textové správy z *SPI*.
- `spi/reader` - Logovacie textové správy z *RFID* čítačky.
- `spi/reader/data/read` - Dáta prečítané z *RFID* čipu spolu s meta dátami v *Json* formáte.
- `spi/reader/state` - *Json* správa so statusom zápisu jednotlivých sektorov.

Koncové body na ktorých obslužná aplikácia čítačky prijíma dáta:

- `spi/reader/data/write` - *Json* správa s dátami pre zápis.

Výpis 4.1: Formát *Json* správy s prečítanými dátami z *RFID* čipu.

```
1 {
2   "uid": [136,4,106,105,143,66,8],
3   "tag": {
4     "tag_vendor": "NXP",
5     "user_memory_offset": 4,
6     "tag_type": "NTAG213",
7     "tag_protocol": 3,
8     "tag_size": 144
9   },
10  "data":
11    [
12    [4,106,105,143],
13    [66,236,76,129],
14    ...
15    [0,0,0,0]
16    ],
17    "read_state": "OK",
18    "timestamp": 1583574457.57088
19 }
```

Prijatá *Json* správa obsahuje jedinečný identifikátor ako pole bytov. Ďalej sa tu nachádzajú meta dáta o čipe, kde `user_memory_offset` je počet stránok pred užívateľskou oblasťou, a `tag_size` je veľkosť užívateľskej pamäte v bytoch. Pod meta

dátami sa nachádzajú už prečítané dáta z celej *RFID* pamäte spolu s časovou známkou a so statusom, s akým sa previedla operácia vyčítania informácií. Dáta majú formát poľa, ktoré obsahuje ďalšie polia a jedno vnútorné pole predstavuje štvor bytovu stránku. Uživateľské dáta začínajú v tomto poli na indexe 4 (pole je indexované od nuly) a končia na indexe, ktorý si dopočítame z veľkosti užívateľskej pamäte a veľkosti jednej stránky. Z uvedeného príkladu nám vyjde, že uživateľská pamäť má 36 stránok a konečný index je 39.

Výpis 4.2: Formát *Json* správy na zápis nových dát do *RFID* čipu.

```
1 {
2   "write_multi":
3   [
4     {
5       "data": [1,3,5,7],
6       "sector": 8
7     },
8     {
9       "data": [2,4,6,8],
10      "sector": 9
11    },
12    ...
13    {
14      "data": [2,4,6,8],
15      "sector": 10
16    }
17  ]
18 }
```

Správa určená na zápis sa skladá z *Json* objektu *write_multi*, ktorý obsahuje pole štruktúr a v jednej štruktúre sa nachádzajú dáta z jedného sektora a číslo sektora. Menšou nevýhodou je, že sa dá zapisovať len po sektoroch (stránkach) o veľkosti štyry byty. Na túto skutočnosť som dbal pri návrhu rozloženia receptúry v pamäti a snažil som sa, aby jedna premenná nebola rozložená cez dve stránky spolu s inou premennou. Napríklad, osem bytový integer, ktorý začína v prvej polovici jednej stránky a končí v polovici druhej stránky. Čísla sektorov v tejto správe nemusia nasledovať za sebou, ale môžu sa preskakovať.

Výpis 4.3: Formát *Json* správy so statusom zápisu sektoru a číslom sektoru.

```
1 {
2   "write": {
3     "sector": 8,
4     "status": "OK"
5   }
6 }
```

Po zápise sa na tému `spi/reader/state` pošle výsledok zápisu pre každú jednu stránku samostatne. Možné statusy sú `OK`, `NOK`.

4.2 Model receptúry

Pri tvorbe modelu receptúry som dbal na veľkosť pamäte, ktorú mám k dispozícii. Tak isto som sa snažil vytvoriť riešenie, ktoré bude nezávislé na type použitého *RFID* čipu.

Kvôli malej veľkosti pamäte sa budú zapisovať len dáta bez názvu parametra. Ďalej by sa dala veľkosť znížiť enkódovaním stringov do `Base64`.

Receptúra sa skladá zo štyroch častí:

- Hlavička
- Kusovník (pole prvkov - indexované od nuly)
- Procedúra (pole procedurálnych krokov - indexované od nuly)
- Stav výroby

V nasledujúcich tabuľkách môžete vidieť popis k jednotlivým prvkom, poradie v akom nasledujú za sebou premenné v pamäti a ich veľkosť v bytoch.

Tab. 4.1: Hlavička receptúry

Názov premennej	Dátový typ *C++*	Popis
RecipeId	uint16_t	Id receptúry
RecipeVersion	uint16_t	Revízia receptúry
ReleaseDateTime	uint64_t	Dátum vydania receptúry
SerialNumber	uint64_t	Sériové číslo produktu
OrderDateTime	uint64_t	Dátum a čas objednávky
DispenseDateTime	uint64_t	Dátum a čas vydania objednávky
Status	uint16_t	Celkový status výroby
PieceListLength	uint8_t	Počet položiek v kusovníku (n)
ProcedureLength	uint8_t	Počet krokov v procedúre (m)

Hlavička má statickú veľkosť 40 bytov. Veľkosť kusovníku a procedúry sa mení v závislosti na počtu prvkov v poli. Jeden prvok v kusovníku má veľkosť 6 bytov a jeden prvok v procedúre má veľkosť 10 bytov. Stavové premenné zaberajú 36 bytov.

Na nasledujúcom obrázku 4.2 je zobrazený *UML* diagram receptúry z aplikácie komunikačného modulu výrobku. Dôležité je, všimnúť si, že triedy hlavička a stav výroby obsahujú *MQTT* klienta. A to z toho dôvodu, že parameter *DispenseDateTime* a *Status* v hlavičke sa zapisujú a zmena sa potrebuje hneď zapísať do *RFID* čipu. To isté platí pre celú triedu `ProductionState`. Parametre, ktoré sa dajú meniť

Tab. 4.2: Kusovník receptúry

Index	Názov premennej	Dátový typ *C++*	Popis
1	Trieda	uint8_t	Trieda materiálu
	Definícia	uint8_t	Druh materiálu
	Množstvo	float	Celkové množstvo
2	Trieda	uint8_t	Trieda materiálu
	Definícia	uint8_t	Druh materiálu
	Množstvo	float	Celkové množstvo
n	Trieda	uint8_t	Trieda materiálu
	Definícia	uint8_t	Druh materiálu
	Množstvo	float	Celkové množstvo

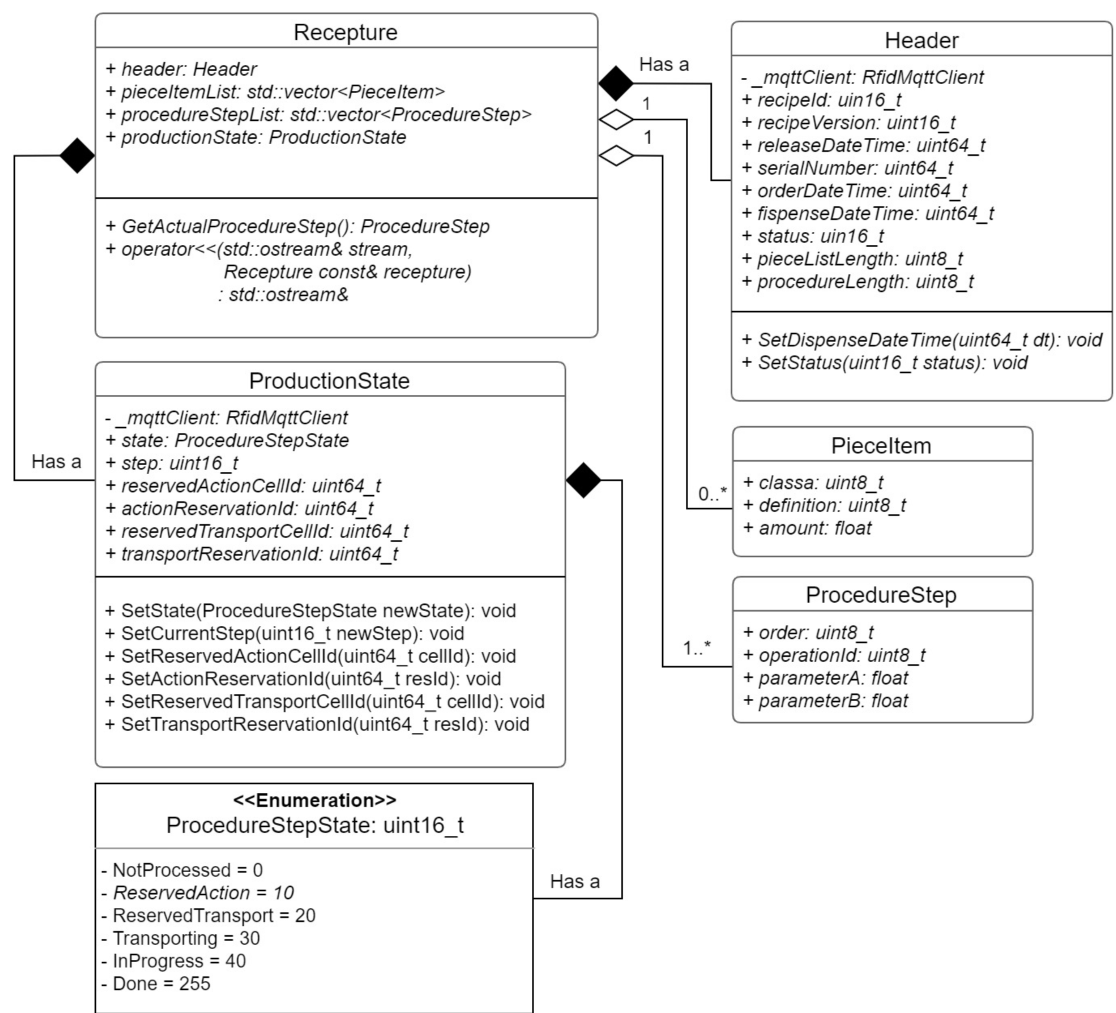
Tab. 4.3: Procedúra receptúry

Index	Názov premennej	Dátový typ *C++*	Popis
1	Poradie	uint8_t	Poradie operácie
	Id Operácie	uint8_t	Id operácie
	Parameter A	float	Prvý vstupný parameter
	Parameter B	float	Druhý vstupný parameter
2	Poradie	uint8_t	Poradie operácie
	Id Operácie	uint8_t	Id operácie
	Parameter A	float	Prvý vstupný parameter
	Parameter B	float	Druhý vstupný parameter
m	Poradie	uint8_t	Poradie operácie
	Id Operácie	uint8_t	Id operácie
	Parameter A	float	Prvý vstupný parameter
	Parameter B	float	Druhý vstupný parameter

Tab. 4.4: Stavové dáta výroby

Názov premennej	Dátový typ *C++*	Popis
State	uint16_t	Stav výrobku
Step	uint16_t	Akt. krok procedúry
ReservedActionCellId	uint64_t	Id rezervovanej bunky
ActionReservationId	uint64_t	Id rezervácie akcie
ReservedTransportCellId	uint64_t	Id rez. transportnej bunky
TransportReservationId	uint64_t	Id rezervácie transportu

majú teda metódy na nastavenie. Vstup je nová hodnota a v metóde sa vytvorí *Json* správa, ktorá sa následne odošle *MQTT* klientom. V prípade úspešného odoslania správy sa nová hodnota zapíše do danej premennej.



Obr. 4.2: *UML* diagram receptúry.

5 Aplikácia

Keďže knižnica obsahujúca implementáciu komunikačného protokolu *OPC UA* je kompatibilná s C99/C++98 prekladačom, rozhodol som sa, že komunikačné moduly procesnej bunky a výrobku budú programované v programovacom jazyku *C++*, konkrétne verziou z roku 2011. Pre vytváranie projektu som si vybral *IDE CLion* od *JetBrains*. Dôvodom bolo hlavne to, že používa na riadenie procesu zostavenia aplikácie *CMake* ako aj knižnica *Open62541*. Toto mi umožní jednoduché pridanie knižnice do projektu. Ďalšou výhodou je, že *CMake* je multiplatformový softvér, ktorý podľa jedného nastavenia vie preložiť aplikácie na rôznych platformách. Konkrétne je celé riešenie kompatibilné s *CMake* verziou *3.10.0* až *3.12.0* . Toto rozmedzie je dané použitými knižnicami.

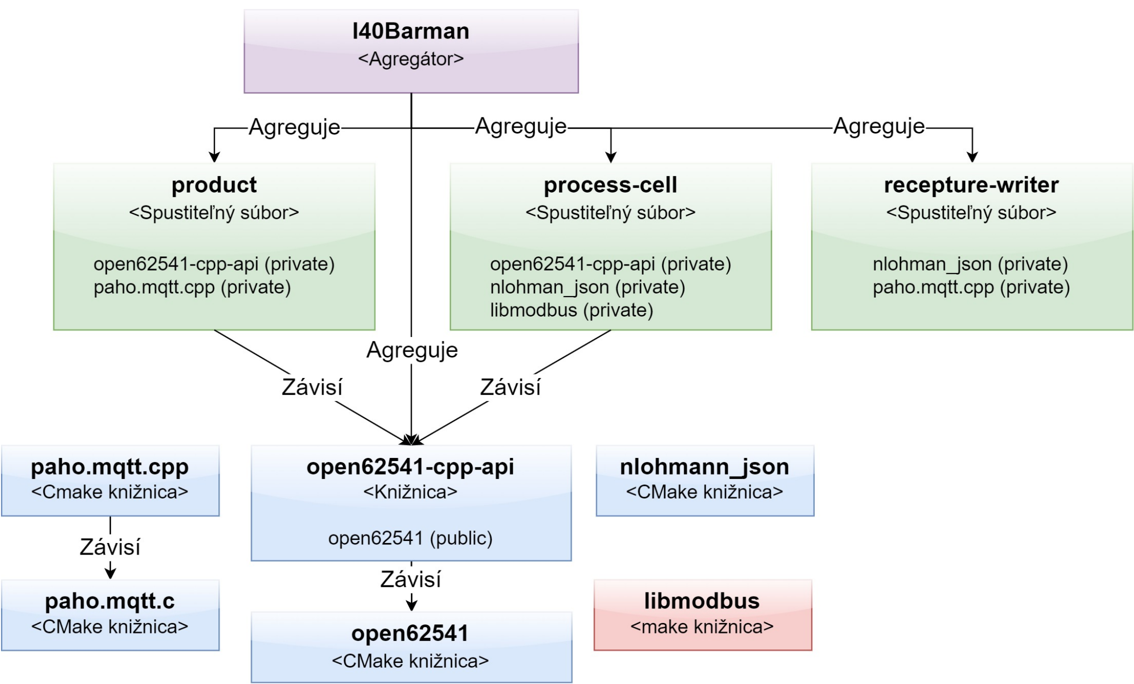
5.1 Štruktúra riešenia

Celé riešenie sa skladá z troch projektov. Dva sú spustiteľné aplikácie, ktoré používajú tretí projekt ako knižnicu. Ďalej som musel vytvoriť dodatočne aplikáciu, pomocou ktorej si nahrám do *RFID* čipu receptúru. Na nasledujúcom obrázku 5.1 môžete vidieť štruktúru projektov a použité knižnice. Popis aplikácii:

- `process-cell` - Aplikácia modulu procesnej bunky.
- `product` - Aplikácia modulu produktu.
- `open62541-cpp-api` - Knižnica, v ktorej zapúzdrujem *Open62541 C* funkcie do *C++* tried a zjednodušujem používanie. Tento projekt je samostatný, aby sa mohol použiť v oboch moduloch.
- `recepture-writer` - Pomocná aplikácia, slúžiaca na zápis receptúry do *RFID* čipu.

Popis knižníc:

- `nlohmann_json` - Knižnica na prácu s *Json*. Používam ju pre načítavanie konfiguračných súborov. Táto knižnica používa tiež na zostavenie *CMake*, čo mi umožnilo jednoduché pridanie do projektu.
- `libmodbus` - Implementácia *Modbus TCP/RTU*. Knižnicu je možné nastaviť, tak, aby pracovala správne na operačnom systéme *Linux* aj *Windows*. Používam ju v komunikačnom module procesnej bunky na komunikáciu s *PLC*.
- `paho.mqtt.cpp` - Implementácia *MQTT* protokolu. Používam ju v komunikačnom module výrobku, kde komunikujem s obslužnou aplikáciou čítačky.
- `open62541` - Implementácia *OPC UA* komunikačného protokolu.



Obr. 5.1: Štruktúra aplikácií.

Každý projekt má svoj `CMakeLists.txt`, čo je vlastne zdrojový súbor pre *CMake*. Na základe nastavení v tomto súbore sa prekladajú zdrojové súbory a vytvára sa výsledná aplikácia. Definuje sa tu verzia *CMake*, typ jazyka, v akom je projekt napísaný a aký prekladač sa má použiť. V rámci projektu sa pridávajú zdrojové súbory, ktoré sa pri zostavovaní majú preložiť. Ďalej sa tu linkujú knižnice tretích strán alebo ďalšie projekty z riešenia. Na obrázku vyššie, majú projekty vypísané, na ktorých projektoch alebo knižniciach závisia. Napríklad projekt `product` závisí na projekte `open62541-cpp-api` s parametrom v zátvorkách, ktorý určuje typ propagácie zdrojových súborov do ďalších projektov. Tento parameter môže nadobúdať hodnoty:

- `PRIVATE` - Zdrojové súbory sú prístupné len projektu, ktorý má nalinkovanú knižnicu a nie projektom vyššie v hierarchii.
- `PUBLIC` - Zdrojové súbory sú prístupné projektu, ktorý má nalinkovanú knižnicu, ale aj ostatným projektom vyššie v hierarchii.
- `INTERFACE` - Táto možnosť sa používa, ak sú z knižnice použité len hlavičkové súbory.

Ako môžete vidieť, projekt `open62541-cpp-api` má nalinkovanú knižnicu *Open62541* ako `PUBLIC`. A to z toho dôvodu, lebo v spustiteľných aplikáciach produktu a bunky používam konštanty a definície z tejto knižnice.

Ďalej môžeme vidieť v projektovej štruktúre komunikačných modulov pri súbore `CMakeLists.txt` súbor s koncovkou `cmake`. Jedná sa o *CMake* skript. V projekte `product` s ním nastavujem správne zostavenie *MQTT* knižnice a v projekte `process-cell` obsahuje pridanie a nastavenie *libmodbus* `make` knižnice.

Všetky aplikácie spravujem pod jedným `git` repozitárom, kvôli jednoduchému manažovaniu. Knižnice sú pridané do projektu ako *git submoduly*. Ak chceme aplikáciu vyvíjať, stačí, ak si naklonujete *git* repozitár pomocou príkazu `git clone <url adresa repozitáru>`. Všetky knižnice tretích strán v projekte sú pridané do riešenia ako *git submoduly*. To má za výhodu to, že sú všetky projekty jednotne nastavené, používa sa vždy tá istá verzia knižnice a nový užívateľ si celé riešenie postaví zo zdrojových kódov po pár príkazoch a nemusí riešiť zložitú inštaláciu knižníc. Následne treba previezť načítanie všetkých knižníc príkazom `git submodule update --init --recursive`. Tento príkaz inicializuje a stiahne správne verzie knižníc tretích strán a dodatok `--recursive` stiahne a inicializuje knižnice práve pridaným knižniciam. A ako posledné treba otvoriť projekt pomocou *CLion*, alebo iného *IDE*, ktoré podporuje stavbu projektov s *CMake*.

Každý projekt má nasledovnú štruktúru:

```
<projekt>
├── icnlude
├── src
├── lib
├── CMakeLists.txt
└── <skript>.cmake (nepovinný)
```

V zložke `icnlude` sa nachádzajú hlavičkové súbory, v `src` zdrojové súbory, v `lib` knižnice pridané pomocou *git* submodulov. A samozrejme *CMake* súbory, podľa ktorých sa zostavuje celé riešenie.

V nasledujúcom výpise kódu sa nachádza ukážka hlavného agregovacieho *CMake* súboru, ktorý zastrešuje celé riešenie. Definuje sa v ňom aká verzia *CMake* môže byť použitá na preklad, meno projektu a verzia *C++* prekladača. Nakoniec sa pridávajú projekty s vlastným *CMake* do celého riešenia.

Výpis 5.1: Ukážka agregovacieho `CMakelists.txt`

```
1 cmake_minimum_required(VERSION 3.10...3.12 FATAL_ERROR)
2 project(I40Barman)
3 set(CMAKE_CXX_STANDARD 11)
4
5 add_subdirectory(open62541-cpp-api)
6 add_subdirectory(process-cell)
7 add_subdirectory(product)
8 add_subdirectory(recepture-writer)
```

V nasledujúcom výpise kódu sa nachádza ukážka projektového *CMake* súboru. Štruktúra všetkých projektov je podobná tomuto. Všeobecne je na začiatku nastavený názov projektu. Za tým nasleduje nastavenie knižníc a pridanie knižníc. Následne pridanie zdrojových a hlavičkových súborov. A nakoniec sa linkujú knižnice k danému projektu. Je dôležité spomenúť, že všetky kižnice sa prekladajú ako statické, čiže po nalinkovaní k projektu vznikne po preklade jeden spustitelný súbor, ktorý obsahuje všetky zdrojové súbory a knižnice v sebe.

Výpis 5.2: Ukážka `CMakelists.txt` z projektu `process-cell`

```
1 project(process-cell)
2
3 # Json library for process configuration file
4 set(JSON_BuildTests OFF CACHE INTERNAL "")
5 add_subdirectory(lib/json)
6 # Setup Libmodbus library
7 include(modbus.cmake)
8
9 # Find all source files with
10 # file extension .c/.cpp inside src folder
11 file(GLOB_RECURSE PROJECT_SRCS "src/*.c" "src/*.cpp")
12 # Add source files to project
13 add_executable(
14     ${PROJECT_NAME}
15     ${PROJECT_SRCS}
16 )
17 # Include project header files
18 target_include_directories(
19     ${PROJECT_NAME}
20     PUBLIC include
21 )
22
23 # Link 3rd party libraries
24 target_link_libraries(
25     ${PROJECT_NAME}
26     PRIVATE open62541-cpp-api
27     PRIVATE nlohmann_json::nlohmann_json
28     PRIVATE modbus
29 )
```

5.2 Knižnica `open62541-cpp-api`

V tejto aplikácii zapúzdrujem *C* kód do *C++* tried, tak, aby sa dalo ľahšie pracovať s *Open62541* a dali sa využiť výhody objektovo orientovaného jazyka. Ďalším dôvodom je, že túto knižnicu používam v dvoch spustiteľných aplikáciach, takže som túto časť kódu oddelil do samostatného projektu, aby sa dala nalinkovať do iných projektov, nie len v rámci mojej práce. V prípade dobrého spracovania *Open62541* knižnice, je možné, že sa táto knižnica začne používať komunitou aj v iných projektoch.

5.2.1 Nastavenie knižnice *Open62541*

Open62541 ponúka nasledovné možnosti zostavenia. Tieto možnosti sú platné k verzii `1.0.1`, ktorá bola vydaná začiatkom roka 2020. Ako bolo už spomínané v sekcii 2.1, táto knižnica je škáľovateľná a je ju možné spustiť na jednoduchých embedded systémoch, až po výkonné počítače či servery. Práve preto sú nasledovné nastavenia veľmi dôležité. Určuje sa v nich, aké *OPC UA* servisy sa vygenerujú, a tým sa škáluje veľkosť aplikácie.
Zoznam všeobecných nastavení [16]:

- **UA_BUILD_EXAMPLES** - Spolu so zdrojovými kódmi sa kompilujú aj príklady.
- **UA_BUILD_UNIT_TESTS** - Spolu so zdrojovými kódmi sa kompilujú aj testy.
- **UA_BUILD_SELFSIGNED_CERTIFICATE** - Vygeneruje sa certifikát, ktorý je určený pre serverovú aplikáciu.
- **UA_ENABLE_AMALGAMATION** - Zo všetkých zdrojových kódov sa vytvorí jeden hlavičkový a zdrojový súbor.
- **UA_ENABLE_COVERAGE** - Zapína funkciu, ktorá spočíta percento pokrytia zdrojového kódu testami.
- **UA_LOGLEVEL** a **MDNSD_LOGLEVEL** - Aká úroveň logovacích správ sa má dostať na štandardný výstup.
  - 600: Fatal
  - 500: Error
  - 400: Warning
  - 300: Info
  - 200: Debug
  - 100: Trace

  Logovacie správy zaberajú veľa pamäte v binárnych súboroch a pri embedded aplikáciach ich nemusí byť treba, preto sa odporúča zvoliť level 600.

Zoznam nastavení, ktorými sa zapínajú *OPC UA* servisy [16]:

- **UA_ENABLE_SUBSCRIPTIONS** - Zapína servis oznámení.
- **UA_ENABLE_SUBSCRIPTIONS_EVENTS** - Zapína servis udalostí (momentálne v experimentálnom mode).
- **UA_ENABLE_METHODCALLS** - Zapína servis *OPC UA* metód.
- **UA_ENABLE_QUERY** - Zapína servis filtrovania adresového priestoru.
- **UA_ENABLE_NODEMANAGEMENT** - Zapína funkciu dynamického pridávania a odoberania uzlov z adresového priestoru počas behu programu.
- **UA_ENABLE_MULTITHREADING** - Zapína funkciu podpory viacej vlákien (momentálne v experimentálnom mode).
- **UA_ENABLE_IMMUTABLE_NODES** - Uzly v adresovom priestore sa needitujú, ale kopírujú a nahrádzajú. Náhrada sa vykoná ako jedna atomická operácia.
- **UA_ENABLE_ENCRYPTION** - Zapína funkciu šifrovaného pripojenia.
- **UA_ENABLE_DISCOVERY** - Zapína servis vyhľadávania serverov (`LDS`).
- **UA_ENABLE_DISCOVERY_MULTICAST** - Zapína servis vyhľadávania serverov s multikastovým rozšírením (`LDS-ME`).
- **UA_ENABLE_DISCOVERY_SEMAPHORE** - Zapína funkciu vyhľadávania serverov so semafórmy.
- **UA_NAMESPACE_ZERO** - Východzí adresový priestor, ktorý obsahuje štandardné definované uzly. Plná definícia, zaberá veľa pamäte a nie je potrebná pre každú aplikáciu. K dispozícii sú tri možnosti[1].
  - **MINIMAL** - Najjednoduchší adresový priestor, ktorý je kompatibilný s väčšinou klientov, ale neprechádza *CTT*[2] testami.
  - **REDUCED** - Jednoduchý adresový priestor, ktorý už prechádza *CTT* testami.
  - **FULL** - Plný adresový priestor generovaný z oficiálnej *XML* definície.

  V prípade vlastného základného adresového priestoru vieme dodať cestu v pokročilých možnostiach do `UA_FILE_NS0`.
- **UA_ENABLE_TYPENAMES** - Do štruktúry `UA_DataType` sa pridá navyše meno dátového typu[1].
- **UA_ENABLE_STATUSCODE_DESCRIPTIONS** - K danému chybovému kódu sa pridá slovný popis[1].

---

[2] *Conformance Testing Tools* definované *OPC Foundation*. Týmito testami musí prejsť každá implementácia *OPC UA* protokolu, ktorá chce dostať oficiálnu certifikáciu.

[1] Táto možnosť zmenšuje výslednú veľkosť aplikácie, a tým umožňuje spúšťať aplikácie aj na embedded systémoch s malou operačnou pamäťou.

V mojom prípade nastavujem vyššie uvedené možnosti pomocou hlavného *CMake* súboru, ktorý patrí danému projektu. Jednotlivé možnosti sú explicitne zapnuté alebo vypnuté. Nastavenie vyzerá nasledovne:

Výpis 5.3: Ukážka nastavení *Open62541* z `CMakelists.txt`

```
1 project(open62541-cpp-api)
2
3 # Set up Open62541 library
4 set(UA_ENABLE_METHODCALLS ON)
5 set(UA_ENABLE_NODEMANAGEMENT ON)
6 set(UA_ENABLE_SUBSCRIPTIONS ON)
7 set(UA_ENABLE_SUBSCRIPTIONS_EVENTS OFF)
8 set(UA_ENABLE_DISCOVERY ON)
9 set(UA_ENABLE_DISCOVERY_MULTICAST ON)
10 set(UA_ENABLE_QUERY OFF)
11 set(UA_ENABLE_ENCRYPTION OFF)
12 set(UA_LOGLEVEL 300)
13 set(MDNSD_LOGLEVEL 300)
14 set(UA_NAMESPACE_ZERO FULL)
15 set(UA_ENABLE_STATUSCODE_DESCRIPTIONS ON)
16 set(UA_ENABLE_TYPENAMES ON)
17
18 add_subdirectory(lib/open62541)
19
20 # Find all source files with file
21 # extension .c/.cpp inside src folder
22 file(GLOB_RECURSE PROJECT_SRCS "src/*.c" "src/*.cpp")
23
24 # Add source files to project
25 add_library(
26     ${PROJECT_NAME}
27     ${PROJECT_SRCS}
28 )
29
30 # Include project header files
31 target_include_directories(
32     ${PROJECT_NAME}
33     PUBLIC include
34 )
35
36 target_link_libraries(
37     ${PROJECT_NAME}
38     PUBLIC open62541::open62541
39 )
```

5.2.2 Štruktúra projektu

V projekte sú zdrojové súbory rozdelené do troch oblastí. Je potrebné spomenúť, že nie všetky funkcie *Open62541* sú implementované, niektoré *C* funkcie, ktorým chýba implementácia sú okomentované. Pri tvorbe tejto knižnice som prešiel cez všetky zdrojové súbory *Open62541* knižnice a vypísal som do príslušných tried hlavičky *C* funkcii. Následne som implementoval len tie, ktoré som potreboval pre moju prácu.

- `Client` - Obsahuje triedy potrebné k vývoju klientských aplikácii. Nachádzajú sa tu triedy:
  - `UaClient` - Trieda zapúzdrujúca funkcionalitu *OPC UA* klienta.
  - `UaSubscription` - Zapúzdrená funkcionalita notifikácii.
  - `UaMonitoredItem` - Pre správnu funkciu notifikácii, tu máme zapúzdrené monitorovacie položky.
- `Core` - Obsahuje spoločné triedy pre klientské aj server aplikácie ako napríklad:
  - `UaNodeId` - Trieda zapúzdrujúca prácu s id uzlov `UA_NodeId`.
  - `UaExpandedNodeId` - Trieda zapúzdrujúca prácu s rozšírenými identifikátormi uzlov `UA_ExpandedNodeId`.
  - `UaVariant` - Trieda zastrešujúca prácu so štruktúrou `UA_Variant`. Jedná sa o generickú triedu, ktorá obsahuje `void*` dáta a dátový typ.
  - `UaNodeContext` - Popis je v sekcii 5.2.3.
  - `UaObjects` - Obsahuje všetky štruktúry definované v *Open62541* zapúzdrené do *C++* tried a pomocné funkcie.
- `Server` - Obsahuje len triedu `UaServer`, ktorá zapúzdruje funkcionalitu *OPC UA* servera.

5.2.3 Trieda `UaNodeContext`

Po vytvorení nových uzlov v adresovom priestore, hlavne premenných, máme len prázdne body, ktoré nám buď vracajú `null` alebo východziu nastavenú hodnotu, ktorá sa už nikdy nezmení. Preto je potrebné jednotlivé premenné napojiť na zdroj, ktorý predstavujú. Napojenie spočíva v nastavení spätných volaní (`callbackov`) a kontextu jednotlivým uzlom. *Open62541* ponúka dva spôsoby napojenia uzla a získanie správnej hodnoty. Prvá možnosť sa nazýva `UA_DataSource` a druhá `UA_Value Callback`. Rozdiel medzi týmito dvoma spôsobmi je ten, že `UA_DataSource` obchádza premennú uzla, v ktorej je uložená hodnota a hneď vracia/zapisuje hodnotu z/do zdroja. Na druhej strane `UA_ValueCallback`, predtým než sa užívateľovi vráti hodnota, ktorú chce prečítať, najprv sa zavolá funkcia `onRead`, kde sa do premennej uzla zapíše nová hodnota a po zapisovaní sa zavolá funkcia `onWrite` a z premennej uzla si vytiahnem novú hodnotu a zapíšem ju do zdroja. Čiže `UA_DataSource` je len spôsob kde sa obchádza interná premenná v uzle.

Výpis 5.4: Štruktúra spätného volania `UA_DataSource` [16].

```
1 typedef struct {
2 UA_StatusCode (*read)
3     (UA_Server *server, const UA_NodeId *sessionId,
4      void *sessionContext, const UA_NodeId *nodeId,
5      void *nodeContext, UA_Boolean includeSourceTimeStamp,
6      const UA_NumericRange *range, UA_DataValue *value);
7
8 UA_StatusCode (*write)
9     (UA_Server *server, const UA_NodeId *sessionId,
10     void *sessionContext, const UA_NodeId *nodeId,
11     void *nodeContext, const UA_NumericRange *range,
12     const UA_DataValue *value);
13 } UA_DataSource;
```

Výpis 5.5: Štruktúra spätného volania `UA_ValueCallback` [16].

```
1 typedef struct {
2 void (*onRead)
3     (UA_Server *server, const UA_NodeId *sessionId,
4     void *sessionContext, const UA_NodeId *nodeid,
5     void *nodeContext, const UA_NumericRange *range,
6     const UA_DataValue *value);
7
8 void (*onWrite)
9     (UA_Server *server, const UA_NodeId *sessionId,
10     void *sessionContext, const UA_NodeId *nodeId,
11     void *nodeContext, const UA_NumericRange *range,
12     const UA_DataValue *data);
13 } UA_ValueCallback;
```

Ďalej napojenie *OPC UA* metódy, spočíva v tom istom. Na to aby metóda fungovala musíme jej priradiť spätné volanie.

Výpis 5.6: Spätné volanie `UA_MethodCallback` [16].

```
1 (*UA_MethodCallback)
2     (UA_Server *server, const UA_NodeId *sessionId,
3      void *sessionContext, const UA_NodeId *methodId,
4      void *methodContext, const UA_NodeId *objectId,
5      void *objectContext, size_t inputSize,
6      const UA_Variant *input, size_t outputSize,
7      UA_Variant *output);
```

Spätné volanie vieme nastaviť aj *OPC UA* definícii dátového typu. Konkrétne nastavíme spätné volanie konštruktoru a deštruktoru premennej alebo objektu.

Výpis 5.7: Štruktúra `UA_NodeTypeLifecycle` [16].

```
1 typedef struct {
2 UA_StatusCode (*constructor)
3         (UA_Server *server,
4          const UA_NodeId *sessionId, void *sessionContext,
5          const UA_NodeId *typeNodeId, void *typeNodeContext,
6          const UA_NodeId *nodeId, void **nodeContext);
7
8 void (*destructor)
9         (UA_Server *server,
10         const UA_NodeId *sessionId, void *sessionContext,
11         const UA_NodeId *typeNodeId, void *typeNodeContext,
12         const UA_NodeId *nodeId, void **nodeContext);
13 } UA_NodeTypeLifecycle;
```

Tieto spätné volania sú súčasťou životného cyklu uzla, ktorý vyzerá nasledovne [16]:

1. Globálny konštruktor (Nastavuje sa v konfigurácii servera)
2. Konštruktor typu uzla.
3. Obdobie používania uzla.
4. Deštruktor typu uzla.
5. Globálny deštruktor (Nastavuje sa v konfigurácii servera)

Problém s týmito spätnými volaniami je ten, že musia byť statické, čiže nie je možné nastaviť nejakému uzlu, `getter` a `setter` nejakej premennej z definície triedy.

Predstavme si situáciu, kde chceme dynamicky vytvoriť v adresovom priestore niekoľko objektov s premennými, ktorý je rovnaký ako *C++* trieda. Táto trieda má v sebe napríklad päť premenných, tri metódy a chceme jej obraz vytvoriť v adresovom priestore. Nastavením statického spätného volania daným premenným, nepomôže, lebo nemáme prístup k premenným danej inštancie triedy. Uzlom, okrem spätných volaní, vieme nastaviť už spomínaný kontext. Je to vlastne zdroj, ktorý daný uzol predstavuje, alebo služba z kadiaľ si vie daný uzol vytiahnuť správnu hodnotu. A k tomuto kontextu máme prístup v daných spätných volaniach, ktoré boli spomínané vyššie.

Kontext vieme nastaviť buď pri vytváraní uzlu, alebo dodatočne pomocou nasledujúcej metódy. Nie vždy vytvárame uzly sami pomocou *API*. Už len z dôvodu, že vytváranie inštancie každého jedného uzla, by zabralo enormne veľa miesta v zdrojových súboroch a takisto by sme nevedeli vytvoriť nové inštancie počas behu programu. Napríklad v prípade vytvárania objektu podľa typu z definície sa jeho premenné a metódy vytvoria samé a my im dodatočne nastavíme kontext.

Výpis 5.8: Metóda `UA_Server_setNodeContext` pre nastavenie kontextu [16].

```
1 UA_StatusCode UA_EXPORT
2 UA_Server_setNodeContext(UA_Server *server,
3                          UA_NodeId nodeId,
4                          void *nodeContext);
```

Daný problém som vyriešil nasledovne. Vyššie spomenuté spätné volania a kontext, ktoré sa dajú nastaviť uzlu som zapúzdril do triedy `UaNodeContext`. Celú triedu môžeme vidieť na *UML* diagrame, ktorý sa nachádza v prílohe A.1. Podčiarknuté riadky znamenajú, že dané premenné alebo metódy z triedy sú statické. Táto trieda je abstraktná, pretože členské metódy su virtuálne a trieda, ktorá ju rozšíri, tieto metódy prepíše. Je to základný blok pre vytváranie vlastných *OPC UA* dátových tipov a dynamické vytváranie *OPC UA* objektov za behu aplikácie.

Uvedieme si príklad ako tento základný blok použiť. Jedná sa o triedu `Management` z aplikácie `process-cell`. Pri vytváraní nového dátového typu treba spraviť nasledovné kroky:

1. Z triedy, ktorá predstavuje daný typ, je nutné zdediť triedu `UaNodeContext`.

```
1 class Management: public UaNodeContext {};
```

2. Vytvoríme statickú metódu `CreateUaObjectType()`, ktorá bude obsahovať definíciu typu.

```
1 static void CreateUaObjectType(UaServer& server) {}
```

- V danej metóde vytvoríme nový typ objektu, ktorý predstavuje triedu a uchováme id uzlu nového typu ako statickú premennú typu `UaNodeId`.

```
1 TYPE_NODEID = UaNodeId(server.GetAppNsIndex(),
2                        "MngmntType");
3 server.AddObjectTypeNode("ManagementType",
4                          Management::TYPE_OBJ,
5                          "Management description.",
6                          TYPE_NODEID);
```

- Danému typu objektu pridáme premenné a metódy, aké sú v triede. Taktiež treba vytvoriť statické premenné typu `string`, ktoré budú obsahovať vyhľadávacie názvy premenných a metód, ktoré sa budú pridávať novému typu. A nakoniec samotné pridávanie, viď príklad. Všimnime si, ako sú použité dopredu vytvorené statické vyhľadávacie názvy `Management::reservationCountVar` a `Management::makeReservationMethod`.

```
1 UaNodeId tmpNodeId=UaNodeId(server.GetAppNsIndex(),
2             Constants::GetUniqueNumericNodeId());
3 server.AddVariableNode("Reservation Count",
4        Management::reservationCountVar,
```

```
5           "Number of successful reservations.",
6           true,
7           UA_ACCESSLEVELMASK_READ,
8           UA_TYPES_INT32,
9           UA_VALUERANK_SCALAR,
10          0,
11          nullptr,
12          TYPE_NODEID,
13          tmpNodeId);
14
15  ArgumentList _inMakeRes;
16  _inMakeRes.AddScalarArgument(
17      "Product Id",
18      "Unique identifier of product.",
19      UA_TYPES[UA_TYPES_UINT64].typeIndex,
20      nullptr);
21
22  _inMakeRes.AddScalarArgument(
23      "Action Id",
24      "Numeric identifier of action.",
25      UA_TYPES[UA_TYPES_BYTE].typeIndex,
26      nullptr);
27
28  _inMakeRes.AddScalarArgument(
29      "Operation Order",
30      "Order of reserved operation in product recipe.",
31      UA_TYPES[UA_TYPES_BYTE].typeIndex,
32      nullptr);
33
34  _inMakeRes.AddScalarArgument(
35      "Parameter A",
36      "Input parameter A for process cell",
37      UA_TYPES[UA_TYPES_FLOAT].typeIndex,
38      nullptr);
39
40  _inMakeRes.AddScalarArgument(
41      "Parameter B",
42      "Input parameter B for process cell",
43      UA_TYPES[UA_TYPES_FLOAT].typeIndex,
44      nullptr);
45
46  ArgumentList _outMakeRes;
47  _outMakeRes.AddScalarArgument(
48      "Status",
49      "True if making reservation was successful, otherwise
           false.", UA_TYPES[UA_TYPES_BOOLEAN].typeIndex,
50      nullptr);
```

```
51
52 _outMakeRes.AddScalarArgument(
53     "Reservation Id",
54     "Unique Id of created reservation.",
55     UA_TYPES[UA_TYPES_UINT64].typeIndex,
56     nullptr);
57
58 tmpNodeId = UaNodeId(server.GetAppNsIndex(),
59                      Constants::GetUniqueNumericNodeId());
60 server.AddMethodNode(
61     "Make Reservation",
62     Management::makeReservationMethod,
63     "Method for create reservation of
64     desired ActionId with parameters.",
65     true,
66     _inMakeRes,
67     _outMakeRes,
68     TYPE_NODEID,
69     tmpNodeId);
```

- Po definovaní všetkých premenných a metód nastavíme novému typu spätné volania na konštruktor a deštruktor.

```
1 SetTypeLifeCycle(server, TYPE_NODEID);
```

3. Ďalej prepíšeme metódu `TypeConstruct` a v prípade potreby aj `TypeDestruct`. V konštruktore dostaneme na vstup identifikátor uzla novej inštancie, ktorý použijeme na hľadanie id uzlov automaticky vytvorených premenných a metód, ktoré nová inštancia obsahuje. Vstupné parametre pre hľadanie sú id uzlu inštancie a uložené statické názvy. S nájdeným id uzlu nastavíme premennej, či metóde kontext a spätné volanie. Po úspešnom prevedení vráti metóda výsledok `true`.

```
1 bool TypeConstruct(
2     UaNodeId& instanceNodeId,
3     UaNodeId& typeNodeId,
4     void* instanceCtx) override {
5     auto foundedNodeId = server.BrowseSimplifiedSingleNodeId
        (
6             instanceNodeId,
7             Management::reservationCountVar,
8             server.GetAppNsIndex());
9     if(!foundedNodeId.IsNull()) {
10         server.SetNodeContext(foundedNodeId, this);
11         SetValueCallback(server, foundedNodeId);
12     }
13
```

```
14     auto foundedNodeId = server.BrowseSimplifiedSingleNodeId
          (
15             instanceNodeId,
16             Management::makeReservationMethod,
17             server.GetAppNsIndex());
18
19     if(!foundedNodeId.IsNull()) {
20         server.SetNodeContext(foundedNodeId, this);
21         server.SetMethodCallback(foundedNodeId,
22                                  MethodCallback);
23     }
24 }
```

4. Prepíšeme aj metódu `ReadValue` a v prípade, že majú premenné povolený zápis aj `WriteValue`. Postup implementácie je rovnaký u oboch. Na vstupe oboch metód je id uzlu, s ktorým sa má pracovať. Podľa daného id uzlu nájdeme meno danej premennej a podľa nájdeného mena priradíme výstupu hodnotu.

```
1 void ReadValue(UaNodeId& node,
2                const UA_NumericRange* range,
3                const UA_DataValue* value) override
4 {
5     std::string name = server.ReadBrowseName(node);
6     UA_Variant val;
7     if(name == Management::reservationCountVar) {
8         UA_Variant_setScalar(&val, &state, &UA_TYPES[
             UA_TYPES_INT32]);
9         UA_Server_writeValue(server.GetServer(), node.
             GetNodeId(), val);
10    }
11 }
```

5. Ďalej prepíšeme metódu `CallBack`, pre správne presmerovanie volania *OPC UA* metódy. Postupujeme tak isto ako v predchádzajúcom prípade a najprv nájdeme názov metódy podľa id uzlu a následne zavoláme metódu z triedy so správnymi vstupmi a výstupmi.

```
1 UA_StatusCode CallBack(
2     UaNodeId& methodNodeId,
3     size_t inputSize,
4     const UA_Variant* input,
5     size_t outputSize,
6     UA_Variant* output) override {
7
8     std::string name = server.ReadBrowseName(methodNodeId);
9
10    if(name == Management::makeReservationMethod) {
11    if(inputSize == 5) {
```

```
12      auto productId = GetVariantValue<uint64_t>(input[0]);
13      auto actionId = GetVariantValue<UA_Byte>(input[1]);
14      auto operationOrder =GetVariantValue<UA_Byte>(input[2]);
15      auto parameterA = GetVariantValue<float>(input[3]);
16      auto parameterB = GetVariantValue<float>(input[4]);
17      // Volanie metody z triedy
18      uint64_t resId = MakeReservation(
19                          productId,
20                          actionId,
21                          operationOrder,
22                          parameterA,
23                          parameterB);
24      UA_Boolean returnVal = true;
25      if(resId == 0) returnVal = false;
26      // Nastavenie vystupnych hodnot
27      UA_Variant_setScalarCopy(
28              output,
29              &returnVal,
30              &UA_TYPES[UA_TYPES_BOOLEAN]);
31      UA_Variant_setScalarCopy(
32              output+1,
33              &resId,
34              &UA_TYPES[UA_TYPES_UINT64]);
35      }
36      else return UA_STATUSCODE_BADMETHODINVALID;
37      }
38 }
```

6. Ako posledné vytvoríme metódu `Instantiate`, ktorá bude vytvárať nové inštancie uzlov. Ako vstupy má metóda uzol rodiča, kde sa má nový uzol pridať, typ referencie a názov novej inštancie.

```
 1 bool Instantiate(UaNodeId& parent,
 2                  UaNodeId& reference,
 3                  std::string name) {
 4 UaNodeId returnNodeId = server.AddObjectNode(
 5                                 name,
 6                                 Management::INSTANCE_OBJ,
 7                                 "",
 8                                 false,
 9                                 parent,
10                                 instanceNodeId,
11                                 reference,
12                                 Management::TYPE_NODEID,
13                                 this);
14
15 return !returnNodeId.IsNull() && instanceNodeId ==
       returnNodeId;
```

```
16 }
```

Ako môžeme vidieť, tento spôsob je založený na znalosti vyhľadávacích mien (`BrowseName`). Jedná sa o vlastnosť uzla, ktorá sa používa pri prehľadávaní adresového priestoru na vytvorenie cesty. Vyhľadávacie meno musí byť jedinečné v rámci objektu [18]. Druhá možnosť definície typov bola spomenutá v kapitole 3.

5.2.4 Pridávanie uzlov v adresovom priestore

Zvolená knižnica *OPC UA* umožňuje pridávať uzle pred a po spustení servera a to aj zo strany klienta. Túto funkcionalitu treba povoliť pri kompilácii ako bolo popísané v sekcii 5.2.1. Pri pridávaní uzlov, treba dbať nato do akého adresového priestoru ich priraďujeme. Štandardne sa v servery nachádza vždy adresový priestor s indexom nula, v ktorom sú definované východzie uzle spolu s informáciami o serveri. V mojom prípade som definoval nový adresový priestor s indexom dva. V tomto adresovom priestore sa nachádzajú všetky uzle patriace procesnej bunke. *Open62541* knižnica má pre každý typ uzla ako vstupné parametre požadované id uzla a vytvorené id uzla.

Prvá možnosť je, vytvárať si vlastné id uzla pomocou počítadla a pri vytváraní nového uzla ho vždy inkrementovať. Toto nové id uzla priradiť na vstup k požadovanému id a vytvorené id uzla sa musí zhodovať s tým na vstupe.

V prípade, že chceme aby vytváranie nového uzla vykonalo *API* stačí, aby požadované id uzla bolo nulové a nastavíme si požadovaný index menného priestoru, inak sa vytvorí uzol vo východzom mennom priestore nula. V nasledujúcej ukážke kódu môžeme vidieť, požadované id uzla `requestedNewNodeId` a vytvorené id uzla `*outNewNodeId`.

Výpis 5.9: Hlavička funkcie pridávania uzla premennej [16].

```
1 UA_Server_addVariableNode(UA_Server *server,
2         const UA_NodeId requestedNewNodeId,
3         const UA_NodeId parentNodeId,
4         const UA_NodeId referenceTypeId,
5         const UA_QualifiedName browseName,
6         const UA_NodeId typeDefinition,
7         const UA_VariableAttributes attr,
8         void *nodeContext,
9         UA_NodeId *outNewNodeId);
```

5.3 Komunikačný modul procesnej bunky - `process-cell`

Cieľom tejto aplikácie je oživiť navrhnutý informačný model bunky z kapitoly 3, ktorý má za úlohu poskytovať potrebné informácie, prijímať rezervácie a spúšťať výrobu. Táto aplikácia mala byť pôvodne na *PLC*, ale keďže to nespĺňa požiadavky *Industry 4.0*, musím dostať procesné dáta o výrobe z *PLC* do aplikácie. Z aplikácie sa konkrétne ovládajú fázy, ktorým sa predávajú vstupné parametre. Snaha bola, aby sa čo najviac dát definovalo z *PLC*, preto sú v ňom definované objekty `Info` a `Manufacruting`. Tieto objekty sa definujú v aplikácii pomocou *csv* súboru, ktorý definuje štruktúru dát a ich adresy, ktoré sa používajú na vyčítanie správnej hodnoty pomocou komunikačného protokolu. Tento súbor je potrebný pre správnu funkciu aplikácie. Podrobnejší spôsob komunikácie, ovládania výroby a mapovania dát je popísaný v sekcii 5.3.2.

Keďže táto aplikácia má byť použitá pre všetky typy buniek, každá aplikácia potrebuje trochu odlišnú konfiguráciu. Ako konfiguračný súbor som zvolil *Json* formát. Bez tohto súboru nie je možné spustiť aplikáciu.

Aplikácia obsahuje dve vlákna, ktoré bežia paralelne. Jedno vlákno obsluhuje komunikáciu s *PLC*, v našom prípade to je *Modbus TCP* klient, ktorý cyklicky číta a zapisuje dáta do *PLC*. Druhé vlákno patrí *OPC UA* serveru. Celé spustenie aplikácie prebieha v nasledujúcich krokoch:

1. Načítanie konfiguračného súboru.
2. Vytvorenie inštancie *OPC UA* serveru s parametrami z konfigurácie.
3. Vytvorenie menného priestoru a dátových typov popisujúcich informačný model bunky.
4. Vytvorenie *OPC UA* objektov `Process-Cell` a `Management` (prepojenie logiky programu s *OPC UA* objektami).
5. Vytvorenie inštancie *Modbus TCP* s parametrami z konfigurácie.
6. Vytvorenie inštancie triedy, ktorá vytvára štruktúru objektov z *PLC* v aplikácií.
7. Vytvorenie štruktúry objektov, ktorá sa komunikuje z *PLC*.
8. Spustenie *Modbus TCP* komunikácie na samostatnom vlákne.
9. Spustenie *OPC UA* servera na samostatnom vlákne.
10. Po vypnutí nastane korektné ukončenie aplikácie a uvoľnenie prostriedkov.

5.3.1 Konfiguračný súbor

Nachádzajú sa tu nastavenia pre *OPC UA* server, ktoré je potrebné nastaviť pri vytváraní inštancie servera a komunikáciu, ktorá slúži na vymieňanie dát s riadiacim systémom. Konfiguračný súbor vyzerá nasledovne:

Výpis 5.10: Konfiguračný súbor aplikácie procesnej bunky.

```
1 {
2   "UaServer": {
3     "Port": 4850,
4     "AppName": "BaverageCell",
5     "AppUri": "urn:vutbr:BarmanI40:BaverageCell",
6     "ProductUri": "http://www.vlada.pl/",
7     "mDnsName": "mDnsBaverageCell",
8     "IpAddress": "192.168.0.221",
9     "Namespace": "http://vutbr.cz/UA/barman/",
10    "Capabilities": [
11      "storage"
12    ]
13  },
14  "Communication": {
15    "CommType": "ModbusTcp",
16    "BuilderType": "S7Csv",
17    "ModbusTcp": {
18      "IpAddress": "192.168.0.5",
19      "Port": 502,
20      "CycleTime": 100,
21      "MinAddress": 0,
22      "MaxAddress": 208
23    },
24    "S7CsvBuilder": {
25      "CsvFileName": "processcell.csv"
26    }
27  }
28 }
```

Pri nastavení servera sa určuje číslo portu a nasledujúce štyri parametre popisujú danú aplikáciu. Za nimi nasleduje nastavenie `IpAddress`, ktoré sa používa na náhradu názvu zariadenia za *IP* adresu zariadenia. Nastavenie `Namespace` sa použije pri vytváraní vlastného menného priestoru. A posledná položka `Capabilities` obsahuje funkcie servera, podľa ktorých komunikačný modul výrobku vyhľadáva servery.

V objekte `Communication` sa určuje typ komunikácie a staviteľa. V prípade neplatnej konfigurácie aplikácia vypíše chybu a ukončí sa. V tomto príklade sa vytvára *Modbus TCP* komunikácia a obsahuje *IP* adresu *PLC* a port, na ktorom je nastavený *Modbus* server. Ďalej tu je doba cyklického načítavania hodnôt a rozmedzie v bytoch, ktoré sa má načítať. V nastavení staviteľa je názov súboru z ktorého sa vytvorí stromová štruktúra objektov.

5.3.2 Spôsob komunikácie a mapovania dát z *PLC* do komunikačného modulu bunky

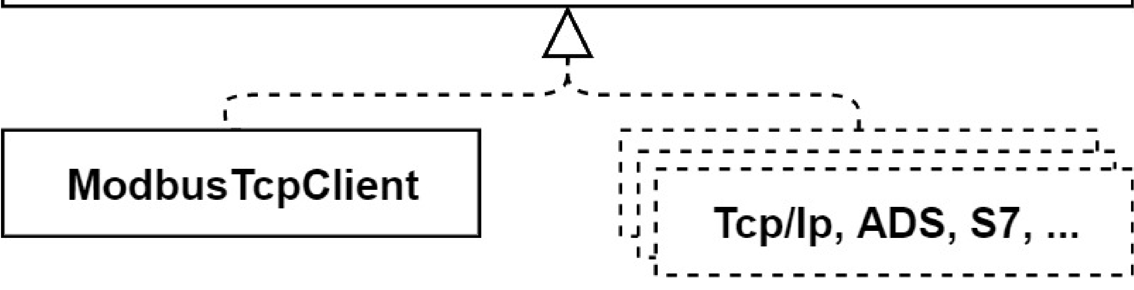
Celou úlohou bolo vymyslieť jednotné rozhranie pomocou ktorého by procesná bunka komunikovala s okolitým svetom. Ako som už spomínal, použité *PLC* nedokáže komunikovať pomocou *OPC UA* a preto je potrebné vytvoriť komunikačné rozhranie mimo *PLC* ako aplikáciu, ktorá bude slúžiť ako brána. Dané rozhranie bolo vytvorené v kapitole 3. Táto aplikácia dáva možnosť komunikovať *PLC* pomocou *OPC UA*, tým že transformuje získané dáta z *PLC* pomocou jedným z komunikačných protokolov, ktoré podporuje.

Pri implementácii komunikácie s *PLC* som sa snažil vytvoriť riešenie, ktoré nebude závisieť od jedného komunikačného protokolu a bude viac univerzálnejšie. Toto som dosiahol vytvorením rozhrania `ICommunication`, ktoré definuje všeobecné hlavičky metód, ktoré sa nevzťahujú k žiadnemu špecifickému komunikačnému protokolu. Pri definovaní nového komunikačného protokolu je potrebné implementovať toto rozhranie. Táto implementácia už bude obsahovať špecifický kód pre daný komunikačný protokol. To akú implementáciu má aplikácia vytvoriť pri spustení je definované v konfiguračnom súbore 5.3.1. Rozhranie `ICommunication` je zobrazené na obrázku 5.2.

Definuje cyklické načítavania dát z *PLC* s periódou `CycleTime`, ktorá je definovaná v konfiguračnom súbore. Takže aplikácia si cyklicky udržuje aktuálne dáta z *PLC* a pri vyčítaní jednej premennej sa aplikácia vlastne dotazuje na tento obraz. Týmto spôsobom urýchlime získavanie jednotlivých hodnôt.

V tomto riešení som implementoval *Modbus TCP* protokol. Dôvod výberu je popísaný v kapitole 2. Na platforme *Siemens Modbus* server používa *IP* adresu zariadenia a na jednom zariadení môže byť spustených viacej serverov naraz. Každý server musí mať nakonfigurovaný odlišný port. Ďalej sa serveru definuje oblasť holdingových registrov ako pointer na dáta blok.

Takže už vieme, že dáta sa budú načítavať do aplikácie cyklicky a poznáme informačný model, ktorý sa má vytvoriť v *OPC UA* adresovom priestore. Teraz je potrebné vyriešiť mapovanie získaných dát do *OPC UA* premenných. Toto mapovanie som vyriešil vytvorením časti informačného modelu v *PLC* pomocou užívateľských definovaných dátových typov (*UDT*). Mapovanie by sa dalo považovať za identické na úrovni objektov a premenných, kde objekty zapúzdrujú premenné, ktoré obsahujú dáta. Jediný rozdiel je pri mapovaní *OPC UA* metódy, ktoré je vysvetlené v samostatnej sekcii 5.3.3. Časť informačného modelu je vytvorená v *PLC* dáta bloku, ktorý je mapovaný do užívateľsky definovanej oblasti (holding registre) *Modbus TCP* servera. Dáta blok, ktorý obsahuje časť informačného modelu spolu s komunikačnými adresami som použil ako predpis pre vytvorenie štruktúry *OPC UA* uzlov v



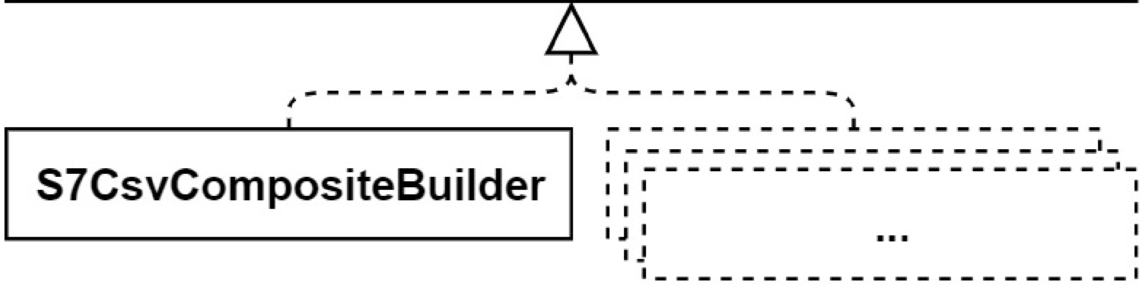
Obr. 5.2: *UML* diagram rozhrania `ICommunicaiton`.

adresovom priestore a vďaka komunikačným adresám som mohol vytvoriť napojenie premenných na správny zdroj dát. Predpis dáta bloku som si exportoval do formátu *csv* jednoduchým označením celého bloku a kopírovaním som preniesol dáta do programu *Microsoft Excel* a ten som uložil ako formát *csv*.

Implementáciu triedy, ktorá tento súbor načíta a daný formát premení na stromovú štruktúru objektov som znova zovšeobecnil a vytvoril rozhranie `ICompositeBuilder`. Stromová štruktúra sa vytvára podľa kompozit návrhového vzoru. Toto rozhranie umožní definovať štruktúru objektov a ich premenných s adresami v rôznych formátoch. Pri vytvorení nového formátu stačí implementovať dané rozhranie. V mojom prípade som implementoval toto rozhranie v triede s názvom `S7CsvCompositeBuilder`. Spracovávanie súboru prebieha riadok po riadku z vrchu nadol. Daná implementácia sa vytvára v aplikácii znova podľa konfiguračného súboru.

Z dáta bloku sa používajú nasledovné údaje, ktoré používam na vytvorenie uzla v adresovom priestore, okrem údaju `Offset`, ktorý používa *Modbus* komunikácia na načítanie správnych dát pre daný uzol:

- *Name* - Používa sa ako názov a vyhľadávací názov uzla.
- *Data type* - Definuje dátový typ uzla, tento údaj sa používa len pri premenných.
- *Offset* - Adresa premennej v dáta bloku. K dátam máme cez *Modbus* prístup



Obr. 5.3: *UML* diagram rozhrania `ICompositeBuilder`.

ako dvoj bytovým registrom, čiže z danej adresy si prepočítam, v ktorom registri sa dáta nachádzajú. Pokiaľ ide o premenné s veľkosťou byte a menej, tak aj pozíciu v registri.

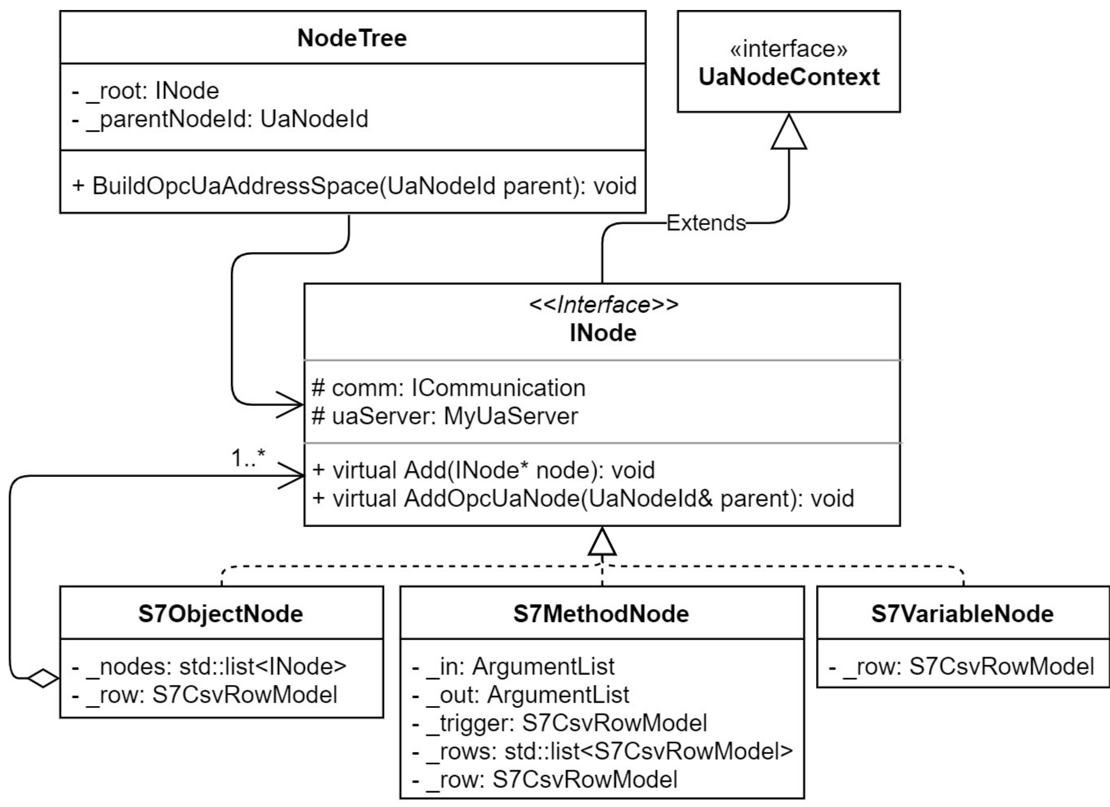
- *Accessible* - Definuje, či je daná premenná čitateľná.
- *Writable* - Definuje či sa do danej premennej dá zapisovať.
- *Visible* - Definuje, či sa premenná má pridať do adresového priestoru.
- *Comment* - Obsahuje viac údajov, ktoré sú oddelené bodkočiarkou. Prvý údaj je typ uzla s úrovňou zanorenia. A druhý údaj sa používa ako popis daného uzla.

Pri exporte dáta bloku sa dajú nepotrebné stĺpčeky schovať. Náhľad *csv* súboru, ktoré používam pre procesnú bunku, sa nachádza v prílohe B.1. Pri spracovaní nastáva problém, keď sme vnorený hlbšie v štruktúre objektov a nasledujúci riadok z *csv* je definícia objektu, ktorý sa nachádza na začiatku stromu. V tomto prípade nemáme ako poznať, že tento nasledujúci objekt patrí na začiatok štruktúry. Preto som musel do stĺpčeku *Comment* pridať číslo, ktoré označuje úroveň zanorenia a k tomu som pridal aj označenie o aký typ uzla sa jedná.

- `o` - Uzol typu objekt.

- `m` - Uzol typu metóda.
- `p` - Uzol typu premenná.

Na nasledujúcom obrázku 5.4 môžeme vidieť *UML* diagram kompozitnej štruktúry. Skladá sa z objektov, ktoré môžu obsahovať niekoľko ďalších objektov, metód alebo premenných. Uzol typu metóda a premenná sú konečné a nemôžu obsahovať ďalšie uzle.



Obr. 5.4: Kompozit návrhový vzor.

Po vytvorení tejto stromovej štruktúry z objektov predám koreňový objekt triede *NodeTree*. A pomocou tejto triedy vytvorím obraz v adresovom priestore *OPC UA* serveru pomocou metódy `BuildOpcUaAddressSpace(UaNodeId parent)`. Táto metóda má ako vstupný parameter id uzla, kde sa má celá štruktúra pridať v adresovom priestore.

Týmto spôsobom dokážem dynamicky previesť rôzne štruktúry z *PLC* do *OPC UA* adresového priestoru. Daný prevod podporuje skoro všetky dátové typy okrem polí. Ako môžeme vidieť daná aplikácia by sa dala použiť aj v iných aplikáciach s inými riadiacimi systémami. Stačí si vytvoriť vlastné implementácie rozhraní `ICommunication` a `ICompositeBuilder`. Alebo môžeme využiť už definovanú komunikáciu *Modbus TCP* a implementovať len nový spôsob vytvárania stromovej štruktúry z iného formátu dát pomocou `ICompositeBuilder` rozhrania.

5.3.3 Mapovanie *OPC UA* metódy a spúšťanie výroby pomocou `RunAction` metódy

Uzol typu metóda v stromovej štruktúre sa z *PLC* mapuje ako objekt, ktorý musí mať v *csv* komentár s označením `m`. Tento objekt má vytvorený v *PLC* špeciálny dátový typ s názvom `GenericMethod` a obsahuje parametre, ktoré sú zobrazené na obrázku 5.5.

GenericMethod		
	Name	Data type
1	RunMethod	Bool
2	ReturnVal	Bool
3	___________	Byte
4	ActionId	USInt
5	ParameterA	Real
6	ParameterB	Real

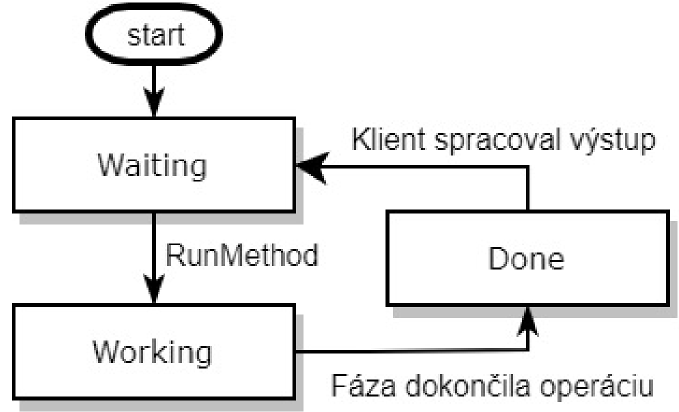
Obr. 5.5: Štruktúra generickej metódy na spúšťanie výroby.

Ak trieda, ktorá skladá stromovú štruktúru narazí na *csv* riadok, ktorý ma v komentári `m`, vytvorí špeciálny koncový uzol stromu a to metódu. Tento uzol má presne dané, že na prvom mieste sa nachádza premenná, s ktorou bude spúšťať akciu. Za touto premennou nasleduje výstupná premenná *OPC UA* metódy, ktorá vráti *true* alebo *false* podľa toho, či je procesná bunka schopná spustiť danú akciu. Ďalej nasledujú vstupy *OPC UA* metódy. V našom prípade to je *ActionId*, podľa ktorého sa spúšťa správna fáza a ďalej parametre pre túto fázu.

Keď sa spúšťa metóda z *OPC UA*, komunikácia najprv zapíše pomocou *Modbus* komunikácie vstupy, následne spustí metódu pomocou premennej `RunMethod` a prečítaný výstup z *Modbus* komunikácie sa nakopíruje do výstupu metódy. Medzi spustením metódy a vyčítaním výstupu je malé oneskorenie, aby sa medzičasom načítali nové dáta z *PLC*.

Funkčný blok v *PLC*, ktorý sa stará o spracovanie *OPC UA* metódy sa nachádza v prílohe B.1. Jeho úlohou je spúšťať operácie pre procesné bunky. Slúži ako jednotné rozhranie pre všetky procesné bunky. Vstupný parameter `ActionId` je použitý preto, aby bolo dané rozhranie viac generické a vedelo spúšťať viac než jednu operáciu. Po spustení metódy sa podľa id akcie vyberie správna operácia a spustí sa so vstupnými parametrami. V ukážkovom príklade z prílohy nespúšťam fázy, ale časovače, s ktorými simulujem vykonávanie práce. Na nasledujúcom obrázku môžeme vidieť stavový automat výroby v procesnej bunke.

Po spustení metódy, kde nastavíme premennú `RunMethod`, prechádzame zo stavu



Obr. 5.6: Stavový automat výroby procesnej bunky.

`Waiting` do `Working`, kde sa spúšťa fáza. Po dokončení prechádzame do stavu `Done` a po spracovaní výrobných údajov komunikačným modulom výrobku prechádza výroba do stavu `Waiting`, kde čaká na nový výrobok.

Na základe opisu funkcie tejto metódy sme schopní vytvoriť ľubovoľné metódy s rôznymi vstupmi. Dôležité je dodržať prvú premennú, ktorá spúšťa metódu a na druhom mieste musí byť jedna výstupná premenná, za ktorou nasleduje ľubovolný počet vstupov. Samozrejme je možné upraviť kód aplikácie tak, aby zvládala viac výstupných premenných. V mojom prípade spúšťam pomocou metódy fázu, ktorej vykonanie trvá nejaký čas a metóda sa musí vykonať za čo najkratšiu dobu. Čiže som nepotreboval viac výstupných hodnôt. Pri implementácii nejakej jednoduchej metódy je možné vrátiť výsledky priamo cez výstupy. V uvedenom príklade si môžeme všimnúť, že dátová štruktúra metódy v *PLC* ma pevne definované vstupy. To sa nám moc nehodí pri vytváraní metódy s inými vstupmi. Naneštastie platforma *Siemens* neponúka možnosť rozšírenia dátových typov. Preto pri implementácii novej metódy je potrebné vytvoriť podľa uvedeného príkladu novú dátovú štruktúru a funkčný blok v ktorom sa zachová stavový automat, ale bude pracovať s inými vstupnými parametrami.

Ako ste si mohli všimnúť na obrázku 5.5, v dátovom type sa nachádza premenná typu *byte*, ktorá nemá čitateľný názov. Pri testovaní aplikácie som narazil na problém, ktorý vznikol kvôli danej implementácii komunikácie a použitým komunikačným protokolom. Problém je v tom, že *Modbus TCP* ponúka možnosťou zapisovať či načítavať do užívateľskej oblasti najmenej jeden register naraz. Tento register má veľkosť dva byty, čiže problém vzniká pri zapisovaní premenných s nepárnou veľkosťou. Konkrétne sa jedná o *bit*, *byte*, *sint* a *usint*. Pri zápise premenných s týmto dátovým typom sa ako prvé načíta z obrazu dáta bloku daný register do ktorého premenná zapadá. Následne sa maskovaním upraví hodnota a celý regis-

ter sa zapíše priamo do *PLC*. A tu nastáva problém ak sa zmenia ostatné hodnoty premenných v danom registry v *PLC* pri zápise z komunikačného modulu bunky sa prepíše celý register so starými hodnotami. Riešením je oddeliť premenné, ktoré sa z *PLC* zapisujú a ktoré len čítajú, do samostatných registrov. Ja som použil na oddelenie premennú, ktorá vyplnila prázdne miesto v registre.

5.3.4 Decentralizované vyhľadávanie procesných buniek v sieti

Ďalšou veľkou funkcionalitou aplikácie je vyhľadávanie všetkých serverov v sieti. Zoznam serverov v sieti nie je centralizovaný, obsahuje ho každá procesná bunka. Na toto používam servis *OPC UA* s názvom `Local discovery server - multicast extension`. Každý komunikačný modul bunky sa správa ako klasický *OPC UA* server, ale aj ako vyhľadávací server (`Discovery`). Multikastové rozšírenie spôsobuje to, že pri štarte aplikácie sa server oznámi, že je v sieti tým, že odošle správu. Táto správa príde každému serveru v sieti. Na základe toho sa server, ktorému príde správa, zaregistruje na tento nový server v sieti. Týmto dosiahneme to, že každá aplikácia procesnej bunky má v sebe zoznam ostatných buniek v sieti.

Nato, aby sme vedeli reagovať na nový server v sieti nám *Open62541* ponúka nastavenie spätného hlásenia. Táto funkcionalita už je naprogramovaná v projekte `open62541-cpp-api` v triede `UaServer`. Tieto spätné hlásenia sa musia nastaviť serveru až po štarte. Okrem toho sa nastavuje aj spätné hlásenie, ak sa nejaký server na danú inštanciu zaregistruje. Nastavovanie vyzerá nasledovne.

Výpis 5.11: Nastavenie spätných hlásenii.

```
10 void SetServerCallbacks() {
11 UA_Server_setServerOnNetworkCallback(
12             _server,
13             ServerOnNetworkCallback,
14             this);
15 UA_Server_setRegisterServerCallback(
16             _server,
17             RegisterServerCallback,
18             this);
19 }
```

V tejto aplikácii je vytvorená trieda `MyUaServer`, ktorá rozširuje `UaServer` z projektu `open62541-cpp-api` a prepisuje metódu `ServerOnNetwork(const UA_ServerOnNetwork* server)`. V tejto prepísanej metóde sa nachádza logika registrovania na oznámený server. Pri registrácii nového servera, je potrebné vytvárať inštanciu *OPC UA* klienta, ktorá prevedie zápis a každý nový oznámený server potrebuje novú inštanciu klienta. Tento problém riešim ukladaním registračného klienta do

hash mapy s registračnou *url* ako kľúčom. Týmto zaistím, aby som sa neregistroval viackrát na ten istý server.

Servery, ktoré sa oznamujú na sieti prídu s vyhľadávacou *url* adresou, ktorá obsahuje hostiteľské meno (`tcp.ip://machine1:4850`) na miesto *IP* adresy (`tcp.ip://192.168.0.152:4850`). Toto zapríčiňuje problém, lebo dané zariadenie, na ktorom beží táto aplikácia nepozná toto meno a nedokážem takúto adresu použiť na registráciu. Preto sa v konfiguračnom súbore nachádza položka `IpAddress`, ktorá sa používa na náhradu hostiteľského mena za *IP* adresu a tým rieši daný problém. V normálnom prípade by sa mal v sieti nachádzať *DNS* server, ktorý by dané hostiteľské mená prekladal na *IP* adresy.

5.4 Komunikačný modul produktu - `product`

Aplikácia slúži na riadenie výroby. Každý výrobok si riadi výrobu sám, čiže v celom riešení je spustených viacej aplikácií naraz. Ako prvé sa vytvára *MQTT* klient podľa konfiguračného súboru, ktorý má formát *Json*. Následne pomocou *MQTT* klienta čakám správu na téme `spi/reader/data/read`. Správa prichádza od obslužnej aplikácie čítačky pri nájdení kontaktu s *RFID* čipom. Následne sa správa, ktorá je dátového typu *string* prevedie do *Json* formátu a následne z *Json* formátu vytvorím objekt receptúry, ktorý je popísaný v sekcii 4.2. Na základe stavu z receptúry sa spustí stavový automat. Celý tento cyklus je naznačený na obrázku v prílohe C.1.

5.4.1 Konfiguračný súbor

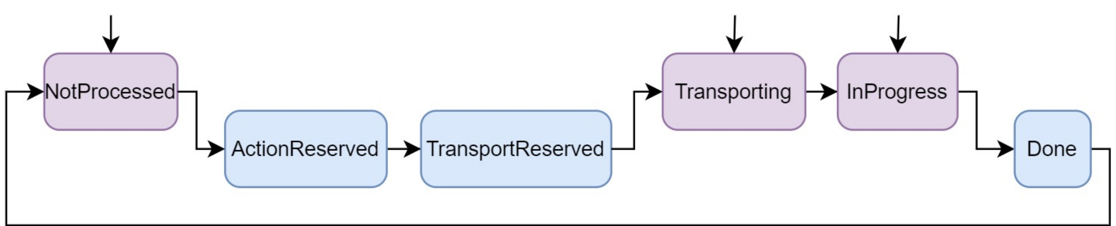
Konfiguračný súbor obsahuje nastavenia pre *MQTT* komunikáciu. *BrokerIpAddress* je adresa *MQTT* prostredníka. *ClientId* identifikátor *MQTT* klienta. *RfidReadDataTopic* téma, na ktorej má *MQTT* čakať správy z čítačky. A *RfidWriteDataTopic* téma, na ktorú aplikácia odosiela správy, keď sa zmenia dáta receptúry v aplikácii. Posledný parameter je nastavenie kvality servisu, doručenia `MQTT` správy, možné hodnoty sú 0, 1 alebo 2.

Výpis 5.12: Konfiguračný súbor aplikácie výrobku.

```
1 {
2   "Mqtt": {
3     "BrokerIpAddress": "tcp://192.168.0.152:1883",
4     "ClientId": "product1",
5     "RfidReadDataTopic": "spi/reader/data/read",
6     "RfidWriteDataTopic": "spi/reader/data/write",
7     "QOS": 1
8   }
9 }
```

5.4.2 Stavový automat výroby

Stavový automat výroby je zobrazený na nasledujúcom obrázku.



Obr. 5.7: Stavový diagram aplikácie.

Fialové stavy sú tie, z ktorých môže začať stavový automat po príchode receptúry z obslužnej aplikácie čítačky. Prejdeme si všetky možnosti, ktoré môžu nastať.

Výrobok sa dostane do stavu `NotProcessed` na začiatku výroby, po nahratí receptúry do výrobku a položenia výrobku na čítačku v sklade pohárov. Tak isto sa dostane do tohto stavu vždy po dokončení akcie v procesnej bunke. Vtedy prejde do tohto stavu zo stavu `Done`.

V stave `NotProcessed` si výrobok podľa operácie, ktorá je na rade nájde všetky bunky, ktoré mu vedia splniť danú operáciu a vyberie si z nich tú, ktorá mu najviac vyhovuje. Vytvorí si notifikáciu na aktuálnu rezerváciu v bunke. A zaregistruje sa pomocou `Management` servisu. Vtedy prechádza výrobok do stavu `ReservedAction`. Keď sa dostane na radu rezervácia, ktorú vytvoril daný výrobok, začnú sa vyhľadávať transportné bunky v sieti. Výrobok si vyberie jednu bunku, nastaví si notifikáciu na aktuálnu rezerváciu a rezervuje si transport. Vtedy sa zmení stav na `TransportReserved`. Po zmene aktuálnej rezervácie si výrobok spustí transport a nastaví sa stav na `Transporting`. Všimnime si, že sú vytvorené spojenia s dvoma bunkami naraz.

Počas presunu stratí výrobok kontakt s čítačkou. Po presune sa spustí stavový automat so začiatkom v stave `Transporting`. Ako prvé sa pripojí aplikácia na transportnú bunku, vyčíta si výsledok akcie prenosu a následne sa pripojí na lokálnu aplikáciu procesnej bunky. Tu skontroluje id bunky, u ktorej má rezerváciu s id aktuálne pripojenou bunkou. Po zhode aplikácia prestaví stav na `InProgress` a spustí akciu na procesnej bunke. V bunkách stratí výrobok kontakt s čítačkou počas vykonávania akcie. V tom prípade po dokončení akcie začne stavový automat od stavu `InProgress`. Aplikácia sa pripojí znova na lokálnu bunku a skontroluje, či sa nachádza výrobná bunka v stave `Done`. Ak áno, nastaví stav výroby tiež na `Done` a skontroluje status, s akým sa dokončila akcia. Ak skončila s chybou nastaví sa celkový status výrobku na `NOK` a zruší sa výroba. V prípade dobrého prevedenia akcie

a ďalších procedurálnych krokov v rade sa inkrementuje aktuálny procedurálny krok a spustí sa stav `NotProcessed`. Celý cyklus výroby sa následne opakuje. Posledná možnosť je, že akcia dopadla dobre, ale bol to posledný krok v receptúre. V tom prípade sa nastaví celkový status výroby na `OK`, nastaví sa čas vydania nápoja a výrobok si nájde transport na výdajné miesto. Po odchode výrobku z procesnej bunky aplikácia zmaže danú rezerváciu a ďalší výrobok má možnosť pokračovať vo výrobe.

5.4.3 Komunikácia s aplikáciou `process-cell`

Komunikácia s procesnou bunkou prebieha prostredníctvom *OPC UA* klienta. V aplikácii som vytvoril triedu `ProcessCellClient`, ktorá predstavuje procesnú bunku a obsahuje *OPC UA* klienta a notifikácie. V triede sú implementované volania metód a načítavanie dát presne také, aké procesná bunka ponúka. Pre túto komunikáciu používam znalosť vyhľadávacích mien, pomocou ktorých si po napojení na procesnú bunku nájdem potrebné id uzlov, ktoré sa používajú na výmenu dát. *UML* diagram triedy sa nachádza v prílohe C.2.

5.4.4 Filtrovanie procesných buniek v sieti

Každý procesný krok obsahuje id operácie ktorá sa má vykonať. Podľa tohoto id si viem filtrovať správne procesné bunky, ktoré mi vedia danú operáciu vykonať. Na filtrovanie používam schopnosti servera (`capabilities`), ktoré má procesná bunka nastavené. Na toto filtrovanie slúži metóda `FindServersOnNetwork(...)`. Použitie môžeme vidieť v nasledujúcej ukážke.

Výpis 5.13: Vyhľadávanie servera pomocou schopností.

```
1 bool FindCell(std::string operation) {
2 StringArray capabilities;
3 ServerOnNetworkArray servers;
4 UA_String capability = UA_String_fromChars(operation.c_str());
5 capabilities.setList(1, &capability);
6
7 bool status = CellUaClient.FindServersOnNetwork(
8                     DISCOVERY_SERVER_ENDPOINT,
9                     0,
10                    0,
11                    capabilities,
12                    servers);
13 if(servers.length() == 0) return false;
14 auto server = ChooseBestProcessCell(servers);
15 ...
16 }
```

Naneštastie táto ukážka kódu nefunguje s *Open62541* verziou `1.0.1`, ktorá je použitá v tomto riešení. Na môj podnet sa táto chyba opravila a oprava bude súčasťou budúceho vydania novej verzie. Z tohto dôvodu som nútený robiť logiku filtrovanie z nájdených serverov v triede `ProcessCellClient`.

5.4.5 Spracovanie notifikácii

Implementácia *Open62541* neponúka vo verzii *1.0.1* samostatné spracovanie požiadaviek notifikácií v *OPC UA* klientovi na pozadí. Pre správnu funkciu je nutné, po nastavení notifikácií na klientovi cyklicky posielať a spracovávať požiadavky pomocou nasledovnej funkcie:

Výpis 5.14: Spracovanie asynchronných požiadavkov a odpovedí [16].

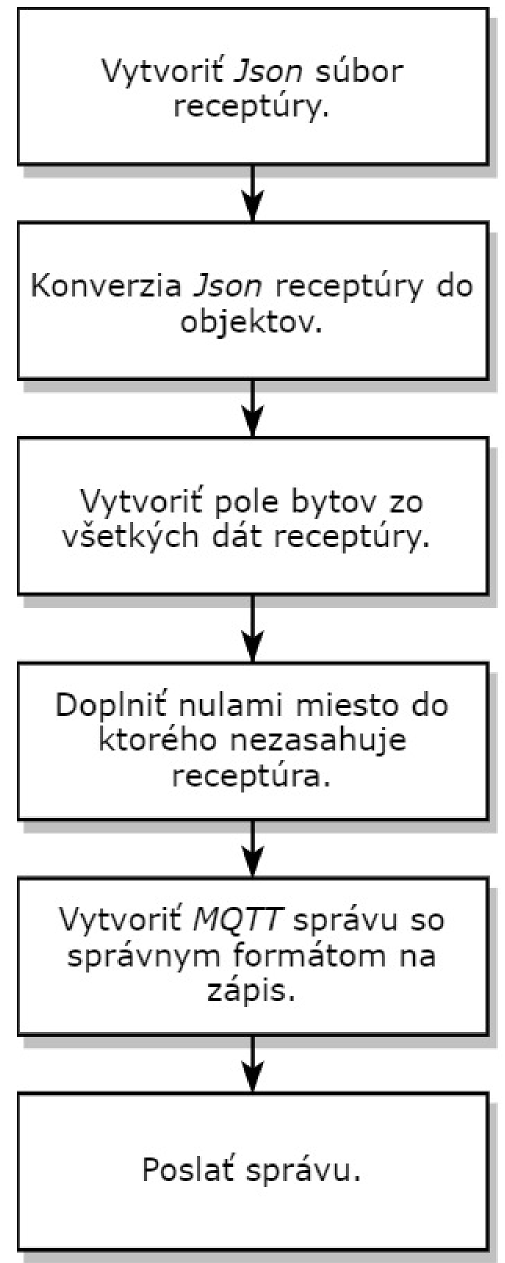
```
1 UA_StatusCode UA_Client_run_iterate(UA_Client* client, UA_UInt16
    timeout);
```

5.5 Zapisovanie receptúry do *RFID* - `recepture-writer`

Funkcia programu je popísaná v krokoch na nasledujúcom obrázku.

Na vstupe je súbor s receptúrou v *Json* formáte. Tento súbor sa načíta a prevedie do príslušného modelu *C++* tried. Ďalej zo všetkých dát spravím vektor s dátovým typom `uint8_t`. Pozor si treba dávať iba pri premenných typu `string`, kde musím vyplniť chýbajúce miesto nulami. Ak je veľkosť receptúry menšia ako veľkosť pamäte čipu, doplním zostávajúce miesto nulami. Tým zaručím, že sa premažú staré dáta. Z výsledného vektoru vytvorím *MQTT* správu v správnom formáte, ktorá sa pošle na `spi/reader/data/write` tému.

Príklad receptúry v *Json* formáte sa nachádza v prílohe D.



Obr. 5.8: Postup aplikácie `recepture-writer`.

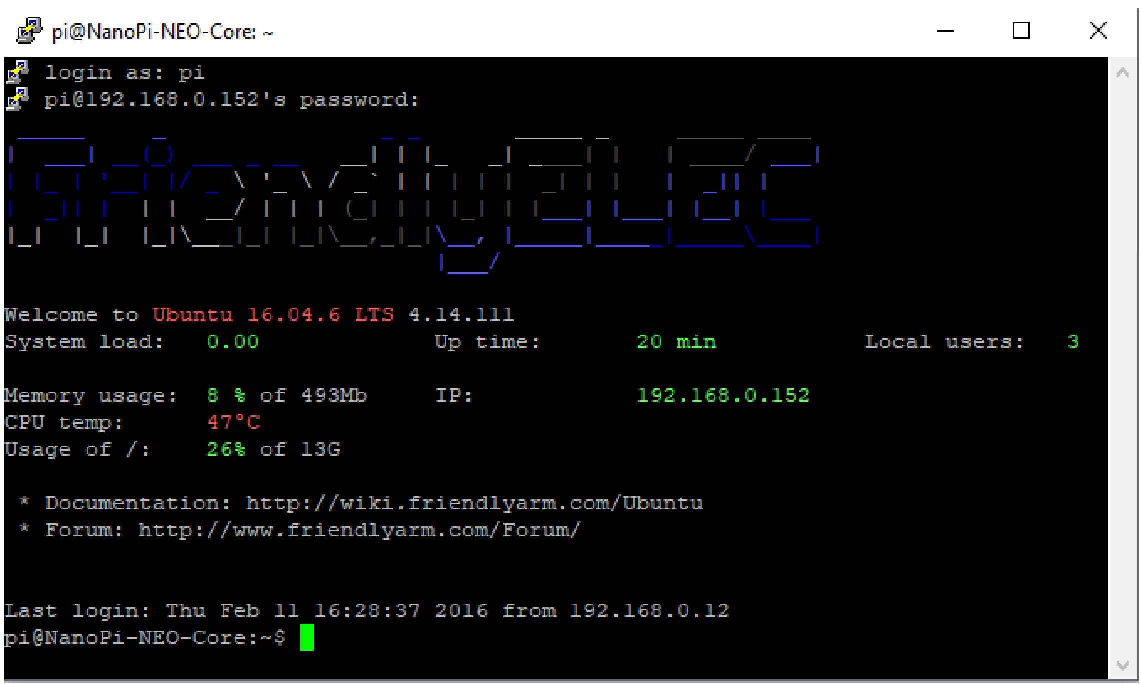
6 Nasadenie aplikácií a testovanie

Popísané aplikácie v predchádzajúcej kapitole budú spustené na *Nano Pi*, na ktorom je špeciálne upravený *Linux Ubuntu 16.04* pre toto embedded zariadenie. Preto je potrebné zdrojové kódy zostaviť priamo na tomto zariadení.

Na prihlásenie sa do systému je potrebné mať pripojené vaše zariadenie spolu s *Nano Pi* do siete so smerovačom, kvôli tomu, že embedded zariadenie má východzie nastavené získavanie *IP* adresy pomocou *DHCP* servera. Po pripojení do siete, by sme mali pomocou nášho smerovača alebo aplikácie, ktorá vyhľadáva zariadenia v sieti nájsť *IP* adresu embedded zariadenia. Túto *IP* adresu použijeme na pripojenie pomocou *SSH* klienta, napríklad *Putty*. Po správnom pripojení treba zadať prihlasovacie údaje:

- `login: pi`
- `heslo: pi`

Po správnom zadaní údajov sa zobrazí nasledovná úvodná obrazovka.



Obr. 6.1: Úvodná obrazovka *Nano Pi*.

Pri tomto návode predpokladám s tým, že v zariadení sa nachádza obslužná aplikácia *RFID* čítačky s názvom `aas-low-level.py` spolu s nainštalovanými balíčkami, ktoré táto aplikácia potrebuje pre správnu funkciu.

Ako prvé prejdeme do súboru, kde sa nachádza táto aplikácia a spustíme ju pomocou nasledovného príkazu.

```
1 $ sudo python3 low-level-spi.py
```

Ďalším krokom je skopírovať z priložených súborov k tejto práci buď priečinok *i40barman* so zdrojovými kódmi alebo už zostavené aplikácie zo zložky *i40barman/bin/Nano Pi* do priečinku *home/pi*. Zostavené aplikácie je treba skopírovať spolu s konfiguračnými súbormi. Po prekopírovaní upravíme konfiguračný súbor podľa popisu v predchádzajúcej kapitole. Aplikácie môžeme spustiť v ľubovoľnom poradí.

6.1 Zostavenie aplikácie zo zdrojových súborov

Ak si chceme aplikáciu zostaviť pre inú platformu alebo sme spravili úpravy v zdrojových súboroch, musíme spustiť zostavenie nasledovne. Ako prvé nainštalujeme nasledovné balíčky pomocou týchto príkazov:

```
1 $ sudo apt-get install git build-essential gcc pkg-config
2     python automake autoconf libtool mosquitto
3     mosquitto-clients
```

Ďalej musíme nainštalovať správnu verziu *CMake*. Celé riešenie je kompatibilné s verziou *3.10.0* až *3.12.0*. Toto rozmedzie je dané použitými knižnicami. Daný *Linux* v *Nano Pi* obsahuje veľmi starú verziu *CMake*, s ktorou zostavenie nie je možné preložiť. Inštaláciu prevedieme nasledovnými príkazmi. Ako prvé odinštalujeme starú verziu *CMake* (ak je nejaká nainštalovaná). A potom stiahneme zdrojové súbory *CMake v3.10.3*, ktoré sa nachádzajú v archíve. Rozbalíme ich a spustíme zostavenie binárnych súborov, ktoré následne nainštalujeme. Ako posledné overíme správnosť inštalovanej verzie príkazom `cmake --version`.

```
1 $ sudo apt purge cmake
2 $ wget http://www.cmake.org/files/v3.10/cmake-3.10.3.tar.gz
3 $ tar -xvzf cmake-3.10.3.tar.gz
4 $ cd cmake-3.10.3/
5 $ ./bootstrap
6 $ make
7 $ sudo make install
8
9
10 $ cmake --version
11 cmake version 3.10.0
12
13 CMake suite maintained and supported by
14 Kitware (kitware.com/cmake).
```

Následne prejdeme do zložky riešenia *i40Barman* a nasledujúcimi príkazmi spustíme zostavenie:

```
1 $/i40Barman mkdir build && cd build
```

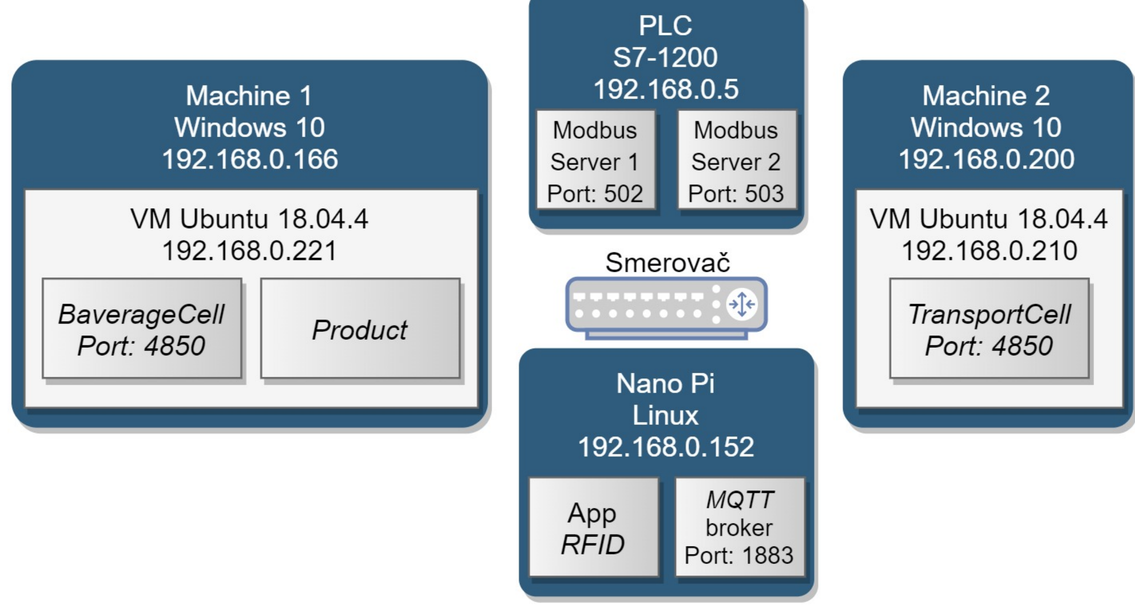
```
2 $/i40Barman sudo cmake ..
3 $/i40Barman sudo make
```

Týmito príkazmi sa nám zostavia všetky projekty v riešení. Zostavené binárne súbory nájdeme v zložke `build`.

6.2 Testovanie distribuovanej výroby

Na testovanie mám k dispozícii len jednu *RFID* čítačku a jedno *PLC*. Na jednej čítačke dokážem spustiť len jeden modul procesnej bunky a výrobku. To preto, že modul bunky spúšťa *OPC UA* server, ktorý prijíma požiadavky na porte 4850. Tento port sa nesmie meniť lebo komunikačný modul výrobku vyhľadáva ďalšie servery práve na tomto porte a dve aplikácie nemôžu využívať ten istý port. S komunikačným modulom výrobku je problém, že prijíma správy z lokálneho *MQTT* brokera, čiže dve inštancie tej istej aplikácie nemá zmysel púšťať na tom istom zariadení. Na jednom programovateľnom logickom automate viem simulovať viacej procesných buniek. Komunikačný modul procesnej bunky komunikuje s *PLC* prostredníctvom komunikácie *Modbus TCP*, na určitom porte. Na jednom automate viem spustiť viac *Modbus* serverov a každý bude komunikovať na vlastnom porte.

Pre testovanie som vytvoril dve procesné bunky. Jednu, ktorá vydáva nápoje s názvom `BaverageCell` a druhá je transportná s názvom `TransportCell`. Komunikačné moduly procesných buniek som spustil na samostatných zariadeniach vo virtuálnom zariadení s operačným systémom *Linux*. Čiže som mal spustené štyri operačné systémy naraz. Všetky zariadenia boli pripojené do spoločnej siete a mali vlastnú *IP* adresu. Komunikačný modul bunky som nespustil na embedded zariadení, aby som nestratil schopnosť krokovania programu. Ďalej som mal na jednom z operačných systémov *Linux* pustený komunikačný modul výrobku, ktorý bol pripojený na *MQTT* prostredníka v *Nano Pi*. Táto aplikácie nebola spustená na *Nano Pi* z toho istého dôvodu. Na embedded zariadení je spustená obslužná aplikácia čítačky spolu s *MQTT* prostredníkom. A na *PLC* sú spustené dva *Modbus* servery, ktoré simulujú procesnú bunku `BaverageCell` a `TransportCell`. Bloková schéma zariadení je zobrazená na nasledujúcom obrázku.

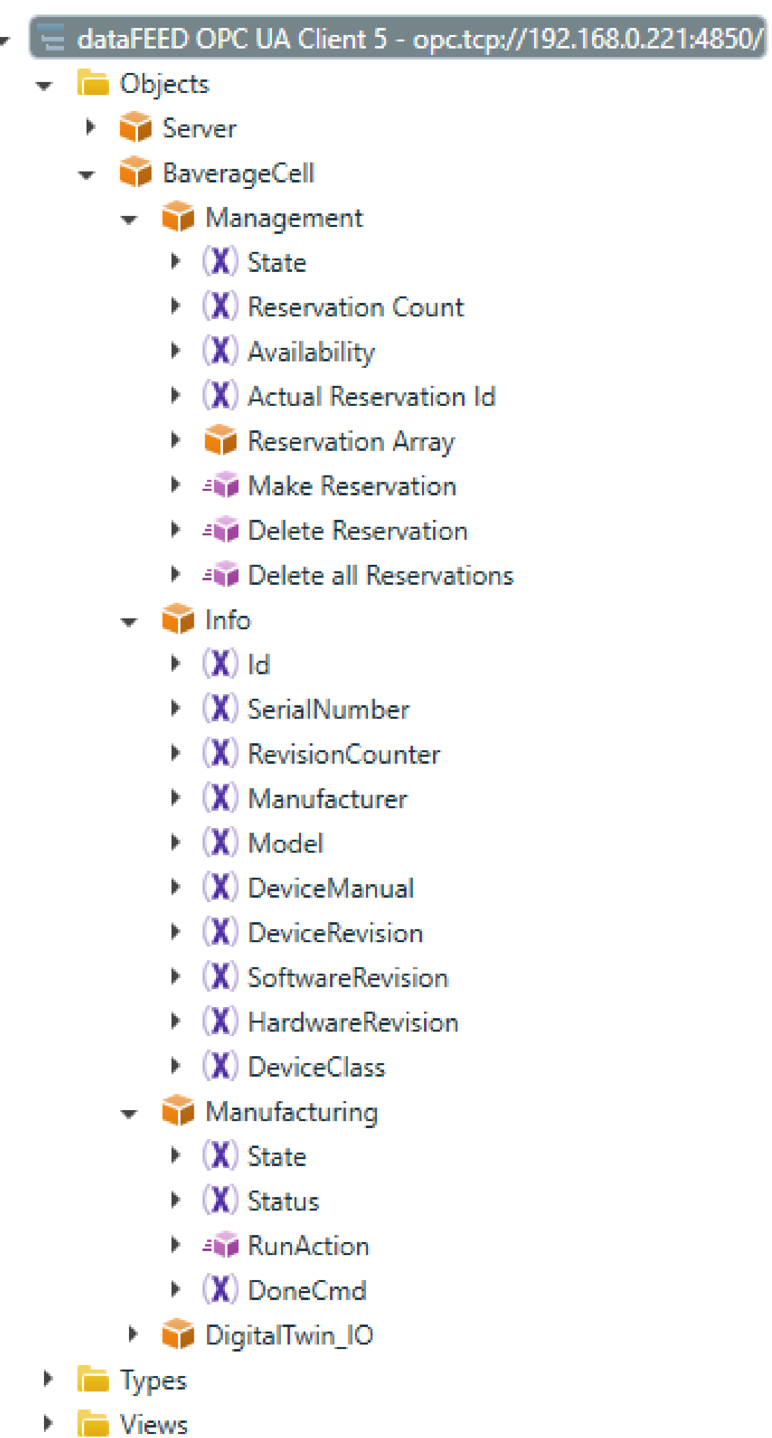


Obr. 6.2: Bloková schéma zariadení pri testovaní.

V tomto teste sa snažím vyrobiť produkt z jednoduchej receptúry, ktorá ma len jeden procedurálny krok. Nachádza sa v prílohe D. Celý výrobný proces by mal zahŕňať nájdenie procesnej bunky, objednanie transportu, vykonanie procedurálneho kroku a objednanie transportu na výdajné miesto. Celý proces výroby som testoval po častiach, vždy od stavu kde prišiel výrobok do kontaktu s čítačkou až po kým ju neopustil. Takto sa celý proces rozdelil na tri časti. Prvá časť prechádza zo stavu `NotProcessed` až do stavu `Transporting`. Vtedy simulujem transport zdvihnutím a znovu položením výrobku na čítačku. Čítačka posiela dáta z *RFID* čipu vždy len pri prvom kontakte. Pošlú sa nové vyčítané dáta do komunikačného modulu výrobku a tento krát sa spustí stavový automat zo stavu `Transporting`. V tomto stave aplikácia skontroluje, či sa dostala na správne miesto a spustí výrobu. Následne sa dostávame do stavu `InProgress` a znovu simulujem vykonanie práce odobratím a znovu položením produktu na čítačku. V tomto stave sa skontroluje výsledok výroby a v prípade, že je to posledný procedurálny krok, tak si výrobok objedná transport na výdaj, inak pokračuje vo výrobe stavom `NotProcessed`.

Pre správne spustenie aplikácie `process-cell` je potrebné mať pri spustiteľnom súbore konfiguračný súbor `cfg.json` a *csv* s názvom, ktorý je nastavený v konfigurácií. Pre správny chod aplikácie `product` stačí mať pri spustiteľnom súbore len konfiguračný súbor.

Na nasledujúcom obrázku môžeme vidieť grafické znázornenie adresového priestoru procesnej bunky `BaverageCell` pomocou *OPC UA* klienta.



Obr. 6.3: Adresový priestor procesnej bunky `BaverageCell` zobrazený pomocou grafického *OPC UA* klienta počas testovania.

7 Záver

Táto diplomová práca sa venovala analýze *test bedu I4.0 Barman* a navrhnutím komunikačného rozhrania medzi výrobkom a procesnými bunkami v distribuovanej výrobe. V prvej kapitole som previedol analýzu test bedu, kde som popísal jednotlivé časti test bedu a na základe toho som určil spôsob distribuovanej výroby. Pri analýze som vychádzal z bakalárskych prác študentov, ktorí pracovali na jednotlivých procesných bunkách a z konzultácií. V ďalšej kapitole som vyberal vhodný komunikačný protokol pre distribuované riadenie podľa konceptu *Industry 4.0*. V analýze som vychádzal zo špecifikácií skupiny *I4.0*, konkrétne z referenčného 3D modelu *RAMI 4.0*, ktorý bol publikovaný v rámci článku o *AAS*. Tu zaviedli komunikačný protokol *OPC UA* hneď v dvoch vrstvách modelu *RAMI*. Konkrétne je použitý ako komunikačný protokol a vďaka jeho sémantickému modelu je použitý aj v reprezentačnej vrstve, kde má úlohu vytvárať digitálny obraz aktíva. Menšia nevýhoda tohto protokolu je, že každé pripojenie klient-server je náročné, či už z časového alebo výpočetného hľadiska. Po vytvorení tohto spojenia si zariadenia môžu spolu vymieňať medzi sebou informácie bezpečne a rýchlo, či už v binárnom tvare, alebo v čitateľnom tvare (*XML*, *Json*), závisí to od použitého transportného protokolu. Po rozbore komunikácií, ktoré ponúka *PLC* v procesných bunkách som usúdil, že nevyhovuje konceptu *I4.0*. Preto komunikačný modul procesnej bunky nebude môcť byť súčasťou automatu, ale je potrebné vytvoriť samostatnú aplikáciu. Následne som spravil rešerš ohľadom výberu implementácie *OPC UA* protokolu. Vyberal som hlavne z aktívnych *OpenSource* implementácií s dobrou dokumentáciou. Ďalším kritériom bolo, aby mala implementovaných čo najviac definovaných servisov *OPC UA*, hlavne tie, ktoré vyplynuli z analýzy. Najdôležitejší servis bol `LDS-ME`, ktorý nám umožní decentralizované vyhľadávanie procesných buniek v sieti. Na základe týchto podmienok sa výber zúžil na *Java* implementáciu *Eclipse Milo* a *C* implementáciu *Open62541*. Druhá uvedená bola aktívnejšie vyvíjaná a spomínaný servis bol plne implementovaný v hlavnej verzii. Práve z týchto dôvodov som vybral túto implementáciu, konkrétne verziu `1.0.1`, ktorá prešla oficiálnou certifikáciou.

V tretej kapitole som vysvetlil dva spôsoby návrhu informačného modelu, buď použitím *API*, ktoré ponúka implementácia *OPC UA*, alebo vytvorením *.xml* súboru, v ktorom sa nachádzajú nové dátové typy a inštancie. Ďalej som sa venoval návrhu informačného modelu procesnej bunky, ktorý som znázornil pomocou oficiálnych grafických symbolov. Pri tomto návrhu som vychádzal z analýzy prevedenej v prvej kapitole. Navrhnutý informačný model obsahuje definície dátových typov. Definícia objektu `Info` obsahuje všeobecné informácie o procesnej bunke podľa oficiálnej špecifikácie modelovania zariadení [17].

V nasledujúcej kapitole som rozobral spôsob výmeny dát medzi *RFID* čítačkou a

komunikačným modulom výrobku, uviedol som formát správy príchodzích dát a dát na zapisovanie. Tak isto som spravil rozbor použitých *RFID* čipov použitých v tomto riešení, hlavne rozloženia pamäte. Ďalšou témou tejto kapitoly bolo definovať formát dát uložených v tejto pamäti. Pri tvorbe som dbal na veľkosť pamäte, ktorú máme k dispozícii. Preto sa do pamäte budú ukladať len dáta bez definície parametrov. Dáta by sa dali rozdeliť do dvoch skupín a to dáta receptúry a stavové dáta z výroby. Pri úprave dát v aplikácii sa zmena hneď zapíše do *RFID* čipu.

V kapitole číslo päť som rozobral celú štruktúru riešenia spolu s použitými knižnicami tretích strán a ich nastavením. V projekt `open62541-cpp-api` som sa zameral na zapúzdrenie *C* knižnice do *C++* tried. V tomto projekte som vytvoril triedu, ktorá je základný blok pre dynamické vytváranie uzlov v adresovom priestore. Následne som tento projekt používal pri vývoji komunikačného modulu výrobku aj bunky. Pri projekte komunikačný modul procesnej bunky som vyriešil mapovanie dát z *PLC S7-1200* pomocou *Modbus TCP*. Vytvorené riešenie dokáže vytvárať pomocou definície dáta bloku z *PLC* uloženej vo formáte *csv*, *OPC UA* objekty, metódy a premenné. Celý spôsob som navrhol, tak, že je možné jednoducho pridať mapovanie dát z iného *PLC* pomocou iného komunikačného protokolu a odlišného formátu dát. Stačí vytvoriť triedu, ktorá implementuje daný interface a pridať nové nastavenie do konfiguračného súbora. Podľa tohto nastavenia sa pri vytváraní aplikácie vytvorí správna inštancia komunikácie. Ďalej som popísal riešenie decentralizovaného vyhľadávania procesných buniek v sieti. Na túto funkciu som použil vlastnosť *OPC UA* komunikačného protokolu. Táto aplikácia tak isto poslúžila na komunikáciu *PLC* s virtuálnym dvojčaťom procesnej bunky. Do adresového priestoru sa pridal nový uzol, ktorý obsahoval vstupy a výstupy virtuálneho dvojčaťa. Pri komunikačnom module výrobku som popísal spôsob komunikácie s procesnou bunkou a priebeh výroby pomocou stavového automatu. Procesné bunky som vyhľadával v sieti s *OPC UA* servisom na základe ich definovaných schopností. Do budúcna by sa mali na *test bede* dorobiť v každej bunke odstavné miesta, kde by mohol výrobok čakať pokiaľ príde na radu v inej procesnej bunke. Týmto by sme zabránili blokovaniu výroby. Ďalej je potrebné určiť chovanie výroby pri určitých chybách komunikácie, ktoré môžu nastať. Pri nasadení tohto riešenia na *test bed Barman* očakávam s prípadnými problémami komunikácie na ktoré som nemohol natrafiť pri provizórnom testovaní. Ako posledné, riešenie obsahuje pomocnú aplikáciu, ktorá slúži na nahrávanie receptúry z *Json* formátu do *RFID* čipu.

V poslednej časti som sa venoval tvorbe manuálu pre správne zostavenie riešenia na operačnom systéme *Linux* a uviedol som výsledky z testovania riešenia.

Pri zavádzaní distribuovaného riadenia si treba uvedomiť, že z pôvodne jedného riadiaceho programu, máme zrazu niekoľko riadiacich programov. To násobne zvyšuje náročnosť spravovania celého systému, už len preto, že pri zmene kódu treba

aktualizovať všetky zariadenia v riešení. Podobne ako pri *test bede Barman*, majú procesné bunky časť kódu rovnakú, je to hlavne kód spoločného rozhrania do ktorého patrí aj komunikácia s obslužnou aplikáciou procesnej bunky. A časť kódu je jedinečná kvôli odlišnému výrobnému procesu. V tomto riešení sa používa platforma *Siemens*, na ktorej sa dá vytvoriť knižnica spoločného kódu. Celé riešenie v *TIA Portále* môže obsahovať viac *PLC* projektov, ktoré by čerpali z tej istej knižnice. Pre vývoj viacerých ľudí na jednom riešení ponúka *TIA Portal* funkciu *multiuser*. Takto by mohli programátori pracovať súčasne na jednotlivých procesných bunkách. Pri tomto spôsobe je otázne, koľko jednotlivých *PLC* projektov by jedno riešenie zvládlo. Poprípade by sa dala preskúmať možnosť, jeden *PLC* program na jedno riešenie a zdielať spoločnú knižnicu naprieč všetkými riešeniami. Dôležité je aby pri zmene spoločnej knižnice v riešení, mali možnosť ostatný členovia týmu synchronizovať si zmeny. Ďalej je tu niekoľko embedded zariadení, na ktorých je spustená obslužná aplikácia čítačky, komunikačný modul výrobku a procesnej bunky. V prípade väčšej továrne by aktualizácia softwaru bola príliš pracná a zdĺhavá. Do budúcna je dobré vymyslieť nejaké automatizované nástroje, ktoré by túto prácu uľahčovali.

Napriek spomenutým mínusom je takýto princíp výroby dôležitý ak chcú spoločnosti vyčnievať na trhu. V dnešnej dobe je veľmi dynamický trh a zákazníci majú stále väčšie požiadavky. So zavedením takejto výroby by sa spoločnosti vedeli rýchlo prispôsobovať požiadavkám trhu za zlomkovú cenu.

Literatúra

[1] Václav KACZMARCZYK, Ondřej BAŠTÁN, Zdeněk BRADÁČ a Jakub ARM. An Industry 4.0 Testbed (Self-Acting Barman): Principles and Design [online]. 2018 [cit. 2020-05-31]. Dostupné z: https://www.sciencedirect.com/science/article/pii/S2405896318309108

[2] Hermann MEISSNER, Rebecca ILSEN a Jan C. AURICH. Analysis of control architectures in the context of Industry 4.0 [online]. 2017 [cit. 2020-05-31]. Dostupné z: https://www.researchgate.net/publication/317048921_Analysis_of_Control_Architectures_in_the_Context_of_Industry_40

[3] EAGAR, MaRi. What is the difference between decentralized and distributed systems? [online]. In: . Nov 4, 2017 [cit. 2020-05-31]. Dostupné z: https://medium.com/distributed-economy/what-is-the-difference-between-decentralized-and-distributed-\systems-f4190a5c6462

[4] Plattform Industrie 4.0. Details of the Asset Administration Shell: Part 1 - The exchange of information between partners in the value chain of Industrie 4.0 (Version 1.0) [online]. 2018 [cit. 2020-05-31]. Dostupné z: https://www.researchgate.net/publication/330142853_Details_of_the_Asset_Administration_Shell_Part_1_-_The_exchange_of_information_between_partners_in_the_value_chain_of_Industrie_40_Version_10

[5] Plattform Industrie 4.0. RAMI4.0 – a reference framework for digitalisation: Reference Architectural Model Industrie 4.0 (RAMI4.0) - An Introduction [online]. 2018 [cit. 2020-05-31]. Dostupné z: https://www.plattform-i40.de/PI40/Redaktion/EN/Downloads/Publikation/rami40-an-introduction.html

[6] Byrtusová Lucie. Výrobní buňka pro testbed Průmyslu 4.0. 2019 [cit. 2020-05-31]

[7] Rejchlík Lukáš. Návrh, konstrukce a programové vybavení inteligentního skladu pro testbed Průmyslu 4.0. 2019 [cit. 2020-05-31]

[8] Horák Lukáš. Návrh, konstrukce a programové vybavení autonomní buňky "Drtič ledu"pro testbed Průmyslu 4.0. 2019 [cit. 2020-05-31]

[9] Dvorský Petr. Návrh, konstrukce a programové vybavení autonomní buňky "Sodovač"pro testbed Průmyslu 4.0. 2019 [cit. 2020-05-31]

[10] Karniš Radim. Návrh, konstrukce a programové vybavení autonomní buňky "Shaker"pro testbed Průmyslu 4.0. 2019 [cit. 2020-05-31]

[11] Sýkora Tomáš. Konstrukce a programové vybavení transportních entit pro testbed Průmysl 4.0. 2019 [cit. 2020-05-31]

[12] OPC Foundation. OPC 11020 - UA Companion Specification Template v1.01.03 [online]. 2019 [cit. 2020-05-31]. Dostupné z: https://opcfoundation.org/developer-tools/specifications-unified-architecture/opc-ua-companion-specification-template/

[13] PROFANTER, Stefan. How to create custom OPC UA Information Models. Https://opcua.rocks/ [online]. April 10, 2019 [cit. 2020-05-31]. Dostupnéz: https://opcua.rocks/custom-information-models/

[14] PROFANTER, Stefan. OPC UA Address Space Explained. Https://opcua.rocks/ [online]. April 15, 2019 [cit. 2020-05-31]. Dostupné z: https://opcua.rocks/address-space/

[15] PROFANTER, Stefan, Kirill DOROFEEV, Alois ZOITL a Alois KNOLL. OPC UA for plug & produce: Automatic device discovery using LDS-ME [online]. 2017 [cit. 2020-05-31]. Dostupné z: https://www.researchgate.net/publication/317358980_OPC_UA_for_plug_produce_Automatic_device_discovery_using_LDS-ME

[16] open62541 authors. Open62541 Documentation: Release 1.0.0-33-g54e2b367 [online]. January 19, 2020 [cit. 2020-05-31]. Dostupné z: https://open62541.org/doc/open62541-1.0.pdf

[17] OPC Foundation. OPC 10000-100: OPC Unified Architecture Part 100: Devices, Release 1.02 [online]. 2019 [cit. 2020-05-31]. Dostupné z: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-100-device-information-model

[18] OPC Foundation. OPC Unified Architecture Specification: Part 3: Address Space Model, Release 1.04 [online]. 2017 [cit. 2020-05-31]. Dostupné z: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-3-address-space-model/

[19] NXP Semiconductors. NTAG213/215/216: NFC Forum Type 2 Tag compliant IC with 144/504/888 bytes user memory [online]. 2015 [cit. 2020-05-31]. Dostupné z: https://www.nxp.com/docs/en/data-sheet/NTAG213_215_216.pdf

Zoznam symbolov, veličín a skratiek

I4.0 Industry 4.0
SCARA Selective Compliance Assembly Robot Arm
ERP Enterprise Resource Planning
AAS Asset Administration Shell
RFID Radio-Frequency Identification
OPC UA OLE for Process Control Unified Architecture
CPPS Cyber-Physical Production Systems
IDE Integrated Development Enviroment
Json JavaScript Object Notation
XML Extensible Markup Language
UML Unified Modeling Language
API Application Programming Interface
PLC Programmable Logic Controller
HMI Human Machine Interface
url Uniform Resource Locator
IP Internet Protocol
DNS Domain Name System
MQTT Message Queuing Telemetry Transport
DHCP Dynamic Host Configuration Protocol
SSH Secure Shell
CTT Compliance Test Tool
LDS-ME Local Discovery Server - Multicast extensions
OOP Object-oriented programming
TCP Transmission Control Protocol
UDP User Datagram Protocol
UDT User-Defined Data Types
csv Comma-separated values

Zoznam príloh

A Knižnica `open62541-cpp-api` **87**

B Aplikácia `process-cell` **88**

C Aplikácia `product` **90**

D Aplikácia `recepture-writer` **92**

E Obsah priloženého CD **93**

A Knižnica `open62541-cpp-api`

UaNodeContext
- _dataSource: UA_DataSource - _valueCallBack: UA_ValueCallback - _nodeTypeLifeCycle: UA_NodeTypeLifecycle - _methodCallback: UA_MethodCallback + instanceNodeId: UaNodeId
+ bool SetTypeLifeCycle(UaServer& server, UaNodeId& typeNodeId); + UA_StatusCode TypeConstructor(UA_Server *server, ... + void TypeDestructor(UA_Server *server, ... + virtual bool TypeConstruct(UaNodeId&, UaNodeId&, void* = nullptr); + virtual bool TypeDestruct(UaNodeId&, UaNodeId&); + bool SetAsDataSource(UaServer& server, UaNodeId& nodeId); + UA_StatusCode ReadDataSource(UA_Server *server, ... + UA_StatusCode WriteDataSource(UA_Server *server, ... + virtual bool ReadData(UaNodeId&, const UA_NumericRange*, UA_DataValue&); + virtual bool WriteData(UaNodeId&, const UA_NumericRange*, const UA_DataValue&); + bool SetValueCallback(UaServer& server, UaNodeId& node); + void ReadValueCallback(UA_Server *server, ... + void WriteValueCallback(UA_Server *server, ... + virtual void ReadValue(UaNodeId&, const UA_NumericRange*, const UA_DataValue*); + virtual void WriteValue(UaNodeId&, const UA_NumericRange*, const UA_DataValue&); + UA_StatusCode MethodCallback(UA_Server *server, ... + virtual UA_StatusCode CallBack(UaNodeId&, ...

Obr. A.1: *UML* diagram `UaNodeContext`.

B Aplikácia `process-cell`

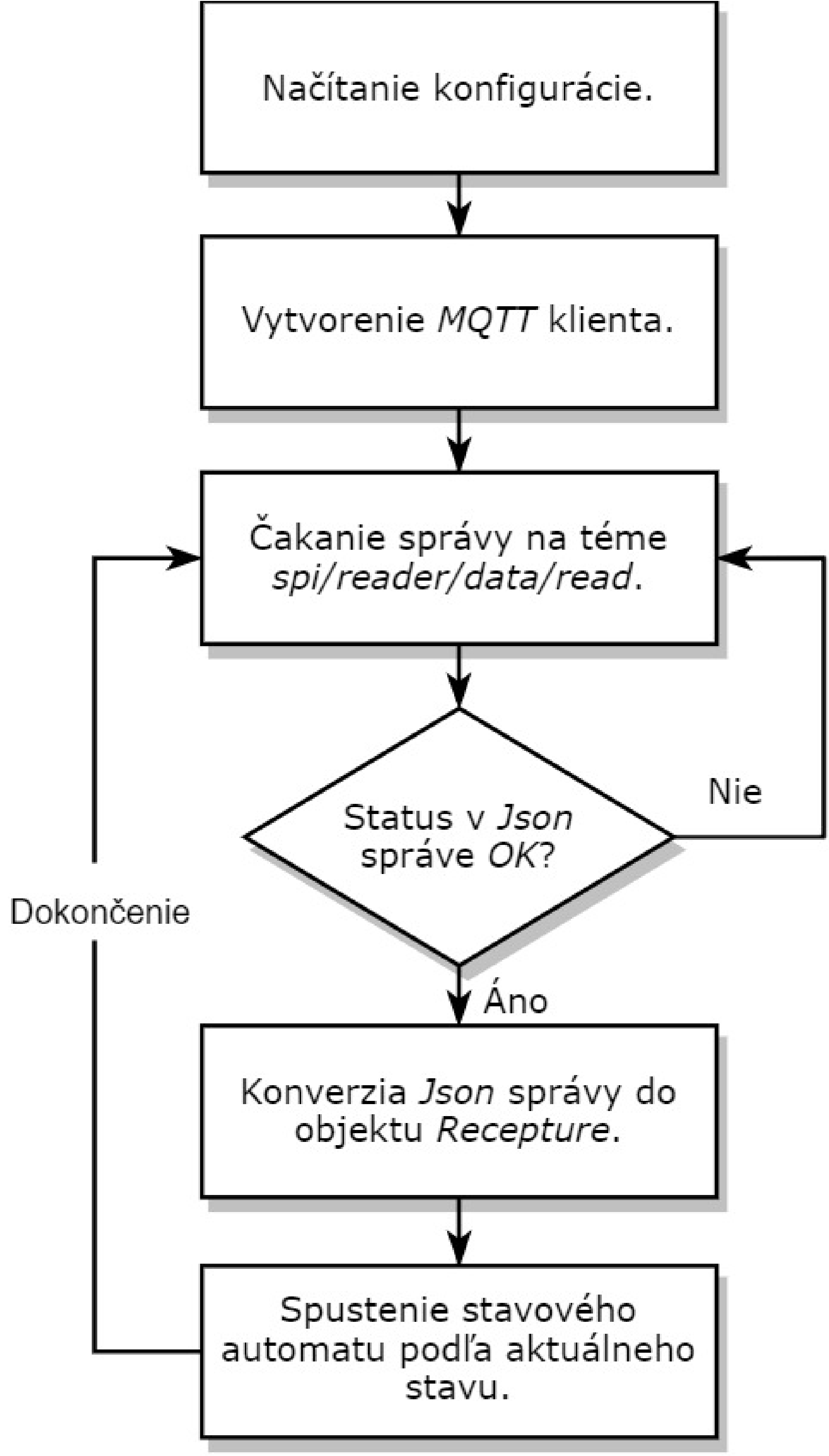
Tab. B.1: Dáta blok exportovaný do formátu *csv*.

Name	Data type	Offset	Accessible	Writable	Comment
Data					o0
Info	InfoNode	0.0	TRUE	FALSE	o1
Id	UDInt	0.0	TRUE	FALSE	p2
SerialNumber	String[20]	4.0	TRUE	FALSE	p2
RevisionCounter	DInt	26.0	TRUE	FALSE	p2
Manufacturer	String[20]	30.0	TRUE	FALSE	p2
Model	String[20]	52.0	TRUE	FALSE	p2
DeviceManual	String[20]	74.0	TRUE	FALSE	p2
DeviceRevision	String[20]	96.0	TRUE	FALSE	p2
SoftwareRevision	String[20]	118.0	TRUE	FALSE	p2
HardwareRevision	String[20]	140.0	TRUE	FALSE	p2
DeviceClass	String[20]	162.0	TRUE	FALSE	p2
Manufacturing	ManufacturingNode	184.0	TRUE	TRUE	o1
State	UInt	184.0	TRUE	TRUE	p2
Status	UInt	186.0	TRUE	TRUE	p2
RunAction	GenericMethod	188.0	TRUE	TRUE	m2
RunMethod	Bool	188.0	TRUE	TRUE	p3
ReturnVal	Bool	188.1	TRUE	FALSE	p3
ActionId	USInt	189.0	TRUE	TRUE	p3
ParameterA	Real	190.0	TRUE	TRUE	p3
ParameterB	Real	194.0	TRUE	TRUE	p3
DoneCmd	Bool	198.0	TRUE	TRUE	p2

```
1
2 REGION States Transitions
3     #RunMethod_RTRIG(CLK := #Manufacturing.RunAction.RunMethod);
4     IF #RunMethod_RTRIG.Q AND #StartConditions THEN
5         #Manufacturing.State := #Working;
6         #Manufacturing.RunAction.ReturnVal := TRUE;
7     END_IF;
8
9
10    IF #RunMethod_RTRIG.Q AND NOT #StartConditions THEN
11        #Manufacturing.RunAction.RunMethod := FALSE;
12        #Manufacturing.RunAction.ReturnVal := FALSE;
13        #Manufacturing.RunAction.ActionId := 0;
14        #Manufacturing.RunAction.ParameterA := 0.0;
15        #Manufacturing.RunAction.ParameterB := 0.0;
16    END_IF;
17
18
19    IF #FakeAction1Delay_TON.Q
20        OR #FakeAction2Delay_TON.Q
21        OR #FakeAction3Delay_TON.Q
22    THEN
23        #Manufacturing.State := #Done;
24        #Manufacturing.Status := #StatusOK;
25
26        #Manufacturing.RunAction.RunMethod := FALSE;
27        #Manufacturing.RunAction.ReturnVal := FALSE;
28        #Manufacturing.RunAction.ActionId := 0;
29        #Manufacturing.RunAction.ParameterA := 0.0;
30        #Manufacturing.RunAction.ParameterB := 0.0;
31    END_IF;
32
33    IF #Manufacturing.DoneCmd
34    THEN
35        #Manufacturing.State := #Waiting;
36        #Manufacturing.Status := #StatusNONE;
37    END_IF;
38    #Manufacturing.DoneCmd := FALSE;
39 END_REGION
40
41 REGION States Logic
42    #FakeAction1Delay_TON(IN := #Manufacturing.State = #Working
43                          AND #Manufacturing.RunAction.ActionId = 1,
44                          PT := T#10s);
45
46    #FakeAction2Delay_TON(IN := #Manufacturing.State = #Working
47                          AND #Manufacturing.RunAction.ActionId = 2,
48                          PT := T#12s);
49
50    #FakeAction3Delay_TON(IN := #Manufacturing.State = #Working
51                          AND #Manufacturing.RunAction.ActionId = 3,
52                          PT := T#14s);
53 END_REGION
```

Obr. B.1: Funkčný blok spracovania *OPC UA* metódy.

C Aplikácia `product`



Obr. C.1: Priebeh aplikácie `product`.

ProcessCellClient

+ cellUaClient: UaClient
+ cellSubscription: UaSubscription
+ CellId: UaNodeId
+ CellInfoId: UaNodeId
+ CellIdId: UaNodeId
+ CellManagementId: UaNodeId
+ CellMakeReservationId: UaNodeId
+ CellDeleteReservationId: UaNodeId
+ CellActualReservationId: UaNodeId
+ CellManufacturingId: UaNodeId
+ CellRunActionId: UaNodeId
+ CellManufacturingStateId: UaNodeId
+ CellManufacturingDoneCmdId: UaNodeId

+ GetManufacturingState(): uint32_t
+ GetManufacturingStatus(): uint32_t
+ WriteManufacturingDoneCmd(): bool
+ ConnectToLocal(): bool
+ FindCell(std::string operation): bool
+ FindNodeIds(): void
+ MakeReservation(uint64_t productId, ...): bool
+ DeleteReservation(uint64_t resId): bool
+ RunAction(UA_Byte actionId, float parA, ...): bool
+ Release(): void

Obr. C.2: *UML* diagram triedy `ProcessCellClient`.

D Aplikácia `recepture-writer`

Výpis D.1: Receptúra v *Json* formáte.


```json
{
  "Header": {
    "RecipeId": 10,
    "RecipeVersion": 1,
    "ReleaseDateTime": 1583446301,
    "SerialNumber": 1000,
    "OrderDateTime": 1583446323,
    "DispenseDateTime": 0,
    "Status": 0,
    "PieceListLength": 1,
    "ProcedureLength": 1
    },
  "PieceItemList":
    [
      {
      "Class":5,
      "Definition":6,
      "Amount":15,
      }
    ],
  "ProcedureStepList":
    [
      {
      "Order": 1,
      "Operation": 1,
      "ParA": 0,
      "ParB": 15,
      }
    ],
  "ProductionState": {
    "State": 0,
    "Step": 0,
    "ReservedActionCellId": 0,
    "ActionReservationId": 0,
    "ReservedTransportCellId": 0,
    "TransportReservationId": 0
  }
}
```

E Obsah priloženého CD

- i40barman .... Zdrojové súbory aplikácií
- ModbusServer_TiaPortal_V14 .... *PLC* program
- Diplomová_práca-Komunikační_rozhraní_pro_testbed_I4.0-Marek_Magáth.pdf
  Elektronická verzia diplomovej práce