VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

DISTRIBUOVANÝ ŘÍDICÍ SYSTÉM S DYNAMICKY MODIFIKOVATELNÝMI UZLY NA PLATFORMĚ RPI ZERO

DISTRIBUTED CONTROL SYSTEM WITH DYNAMICALLY EVOLVABLE NODES ON RPI ZERO PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MICHAL SZYMIK**
AUTHOR

VEDOUCÍ PRÁCE — **Doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.**
SUPERVISOR

BRNO 2021

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Szymik Michal**
Program: Informační technologie
Název: **Distribuovaný řídicí systém s dynamicky modifikovatelnými uzly na platformě RPi Zero**
Distributed Control System with Dynamically Evolvable Nodes on RPi Zero Platform
Kategorie: Operační systémy
Zadání:

1. Prostudujte problematiku distribuovaných řídicích systémů (DCS), internetu věcí (IoT) a systémů dispečerského řízení a sběru dat (SCADA).
2. Seznamte se s existující pokusnou implementací operačního systému pro rekonfigurovatelný IoT uzel v jazyce Python na platformách ESP32 a Raspbery Pi, komunikující protokolem MQTT. Také se seznamte s možností programovat mikroaplikace na uzlech vysokoúrovňovými Petriho sítěmi.
3. Na základě konzultace s vedoucím práce navrhněte celkovou architekturu DCS s možností dynamicky modifikovat software na uzlech DCS prostřednictvím MQTT. Zaměřte se na operační systém uzlů, databázi, uživatelské rozhraní (HMI) a prostředky pro monitorování a správu systému.
4. Navržený systém realizujte s využitím existujících i vámi nově implementovaných komponent, proveďte testování za pomoci vhodné aplikace a vyhodnoťte dosažené výsledky.

Literatura:

- Drahovský, P.: Rekonfigurovatelný IoT uzel na bázi ESP8266/ESP32. Bakalářská práce FIT VUT v Brně. URL: https://wis.fit.vutbr.cz/FIT/db/dir.php/rp/2017/BP/20280.pdf
- Pitko, E.: IoT s rekonfigurovatelnými uzly v Pythonu. Bakalářská práce FIT VUT v Brně. URL: https://wis.fit.vutbr.cz/FIT/db/dir.php/rp/2018/BP/21600.pdf
- Grigorev, D.: Interpret vysokoúrovňových Petriho sítí v Pythonu. Bakalářská práce FIT VUT v Brně. URL: https://wis.fit.vutbr.cz/FIT/db/dir.php/rp/2018/BP/21634.pdf
- SNAKES. URL: https://www.ibisc.univ-evry.fr/~fpommereau/SNAKES/

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a část návrhu.

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

Vedoucí práce: **Janoušek Vladimír, doc. Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Tato práce je zaměřena na tvorbu distribuovaného řídicího systému, běžícího na zařízeních Raspberry Pi Zero, s možností jeho dynamické rekonfigurace. Řešení tohoto problému je implementováno v jazyce Python ve formě běhového prostředí – runtime platformy. Práce dále popisuje tvorbu aplikací pro vytvořený systém a představuje skript, který slouží k usnadnění instalace aplikace na uzly systému. Vytvořený systém je funkční a je demonstrován na ukázkové aplikaci.

Abstract

This work focuses on the development of a distributed control system that runs on multiple Raspberry Pi Zero devices and enables dynamic reconfiguration of its topology. The platform is implemented in Python in the form of a runtime environment. This work also describes how to develop applications for this system and introduces a script that facilitates the installation of an application onto the system nodes. The system described works and a sample application demonstrates its functionality.

Klíčová slova

Raspberry Pi Zero, Python, MQTT, DCS, PLC, SCADA, Node-RED

Keywords

Raspberry Pi Zero, Python, MQTT, DCS, PLC, SCADA, Node-RED

Citace

SZYMIK, Michal. *Distribuovaný řídicí systém s dynamicky modifikovatelnými uzly na platformě RPi Zero*. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. Vladimír Janoušek, Ph.D.

Distribuovaný řídicí systém s dynamicky modifikovatelnými uzly na platformě RPi Zero

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Doc. Ing. Vladimíra Janouška Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

....
Michal Szymik
11. května 2021

Poděkování

Rád bych touto cestou poděkoval Doc. Ing. Vladimíru Janouškovi Ph.D. za veškeré jeho rady a vedení, které mi poskytl během tvorby této práce.

Obsah

1 Úvod **3**

2 Distribuované řídicí systémy **4**
2.1 Úvod do distribuovaných řídicích systémů . . . 4
2.1.1 Architektura distribuovaných řídicích systémů . . . 5
2.2 SCADA . . . 5
2.2.1 Struktura SCADA systémů . . . 6
2.2.2 Komunikační systém . . . 6
2.3 PLC . . . 7
2.3.1 Programování . . . 7
2.3.2 Diagram funkčních bloků . . . 8
2.3.3 Strukturovaný text . . . 9
2.3.4 Rozdíl mezi DCS a SCADA . . . 10
2.4 Komunikační systém . . . 10
2.4.1 MQTT . . . 10
2.4.2 Vlastnosti MQTT . . . 10
2.4.3 Princip MQTT . . . 10
2.4.4 Implementace MQTT . . . 11

3 Použitý hardware a software **12**
3.1 Raspberry Pi . . . 12
3.2 Software . . . 14

4 Analýza a návrh systému **15**
4.1 Distribuovaný řídicí systém . . . 15
4.2 Uzly DCS . . . 15
4.3 Uživatelská aplikace . . . 16
4.4 Funkční blok . . . 17
4.4.1 Dělení funkčních bloků . . . 17
4.5 Komunikace DCS . . . 18
4.5.1 Dynamická rekonfigurace . . . 24
4.6 Runtime platforma . . . 24

5 Implementace **27**
5.1 Použité knihovny . . . 27
5.2 Runtime platforma . . . 27
5.2.1 Modul konfigurace . . . 27
5.2.2 Modul klient . . . 29

5.2.3 Modul HMI . . . 29
5.2.4 Hlavní modul . . . 30
5.3 Inicializační skript . . . 34

6 Tvorba aplikací pro realizovanou platformu a její testování 35
6.1 Tvorba aplikace . . . 35
6.2 Demo aplikace . . . 37
6.2.1 Popis problému . . . 37
6.2.2 Hardware využitý v demo aplikaci . . . 39
6.2.3 Implementace demo aplikace . . . 41
6.2.4 Node-RED . . . 43

7 Závěr 47

Literatura 48

A Obsah přiloženého paměťového média 50

B Další vývoj práce 51

Kapitola 1

Úvod

V době neustálého rozvoje automatizace průmyslových výrobních procesů, chytrých domácností a dalších podobných systémů, využívajících ke svému účelu různých druhů chytrých zařízení, se řeší problémy spojené s tím, jak vytvořit systém, který by dokázal efektivně realizovat libovolné zadání a byl by schopný jej v průběhu svého běhu měnit, aniž by došlo k jeho odstavení. Jedny z možných zařízení, ze kterých by se mohl tento systém skládat, jsou počítače velmi malých rozměrů - Raspberry Pi Zero. Tato zařízení jsou schopná vykonávat stejné funkce jako klasické počítače (i když pochopitelně s omezenějším výkonem) a zároveň jsou díky své menší velikostí ideální jako řídící prvky v systému, na který jsou kladeny nároky, aby byl navenek co nejméně viditelný.

Řídicí systémy, které jsou na světě už delší dobu, procházely od svého vzniku značným vývojem, který je motivovaný stále většími nároky na jeho funkčnost, ale také neustálým rozvojem technologií, kterých systémy využívají. Zářným příkladem je například vznik komunikačních protokolů umožňujících bezdrátové propojení prvků systému. S příchodem chytrých zařízení, jako je třeba výše uvedený počítač Raspberry Pi, se pro vývoj řídicích systémů otevírá spousta nových možností. „Inteligence" prvků systému umožňuje rozložit jeho řízení a tím se zbavit některých nepříjemných vlastností systému, které jsou řízeny jedním centrálním prvkem. Mou motivací pro napsání této práce je vytvoření takového systému, který využívá počítačů Raspberry Pi Zero a je schopný díky rozložení řízení modifikovat činnost jednotlivých prvků dynamicky za jeho běhu.

Tato práce se věnuje návrhu a realizaci výše uvedeného systému. První kapitoly textu se zabývají teorií existujících systémů a technologií, které je možné v řídicích systémech používat. Čtenář se tak dozví všechny potřebné informace, ze kterých se pak dále vychází v druhé části práce, kde je popisován samotný systém, jeho analýza, návrh a způsob, kterým byl implementován. Poslední část se věnuje ukázce systému na demo aplikaci. V závěrečné kapitole se pak nachází shrnutí celé této práce, včetně nejdůležitějších poznatků, ke kterým jsem při práci na tomto projektu došel.

Kapitola 2

Distribuované řídicí systémy

Předmětem této bakalářské práce je téma distribuovaných řídicích systémů. V této kapitole se proto budu věnovat jak jejich popisu, tak popisu částí, ze kterých se skládají. Tato kapitola také obsahuje popis systémů SCADA[1], které s distribuovanými řídicími systémy úzce souvisí a které využívá PLC[2] programování, kterým jsem se značně inspiroval při návrhu svého vlastního systému.

2.1 Úvod do distribuovaných řídicích systémů

Distribuovaný řídicí systém, anglicky Distributed Control System (dále jen DCS), je speciální řídicí systém, který se skládá z více různě rozmístěných prvků, jež řídí jeho činnost. O prvcích těchto systému se hovoří jako o jeho uzlech, které řídí jednotlivé procesy, zařízení či skupiny zařízení. Vzájemné propojení uzlů tvoří síť celého systému.

Získávání dat a řízení funkcí je umožněno díky mikroprocesorové podstatě uzlů systému, které se nachází v blízkosti míst, kde sběr dat a provádění funkcí probíhá. Uzly jsou schopné komunikovat mezi sebou a s případnými dalšími operačními či řídícími zařízeními. Uzly jsou napojeny na periferní zařízení jako jsou sensory a aktivátory. Pro propojení můžou být použity různé typy sběrnic a stejně tak i různé typy přenosových protokolů. DCS jsou vhodné pro použití v rozsáhlých výrobních továrnách, které se skládají z většího množství řídicích smyček. [4]

Distribuovanost DCS spočívá v rozložení řízení systému do menších subsystému, které fungují nezávisle na ostatních částech. To vede k větší flexibilitě a spolehlivosti systému, díky čemu výpadek jednoho subsystému nezpůsobí výpadek celého systému.

DCS umožňují implementovat komplexní řídící strategie, které vedou k automatizaci procesů, které systém řídí.

[1]SCADA – Supervisory Control And Data Acquisition
[2]PLC – Programmable Logic Controller

2.1.1 Architektura distribuovaných řídicích systémů

Mezi základní komponenty DCS patří kontrolní, řídící a operační stanice, HMI[3], jednotky řízení procesů, chytrá zařízení a komunikační systém.

Operační stanice V DCS hraje roli centralizovaného operačního řídicího centra, které slouží k sledování celého systému. Umožňuje monitorování operací, produkce, různých událostí a chyb.

HMI Jedná se o rozhraní mezi člověkem a zařízením. Funkcí HMI v DCS umožňuje operátorovi systému sledovat stav či hodnoty jednotlivých zařízení v systému a jejich řízení. Pokud je pro tyto funkce použit počítač, pak se hovoří o Human-Computer Interaction[4]. Možnosti HMI se liší podle potřeb a zaměření DCS. Komplexnost HMI se přizpůsobuje prostředí, ve kterém se využívá.[5]

Jednotky řízení procesů Jedná se o zařízení s vetším výpočetním výkonem, který je různými typy vstupně/výstupních portů propojený s periferními zařízeními, a to přímo nebo vzdáleně. Tyto jednotky sbírají data ze svých vstupů, následně je analyzují a zpracovávají podle řídicí logiky, která je v nich implementována, a posílají výstupy na své výstupní porty, které řídí aktivátory. Typickým představitelem těchto jednotek jsou PLC a RTU zařízení.

Komunikační systém Hlavní rolí komunikačního systému v DCS je umožnit propojení řídicích stanic, řídicích jednotek a periferních zařízení. To umožňuje přenášení informací. Komunikace může být řízena jedním čí více různými komunikačními protokoly. Mezi běžně používané patří Ethernet, Profibus, Foundation Field Bus, DeviceNet, Modbus, atd. Různé protokoly můžou být vhodné pro různé úrovně systému. Pro kritické úrovně se používá redundantní propojení, aby v případě výpadku jedné komunikační sběrnice nebyla ohrožena činnost systému.

2.2 SCADA

SCADA, neboli Dispečerské řízení a sběr dat[5] (dále jen SCADA), je řídicí systém, skládající se ze softwarových a hardwarových komponent, jehož činnost je zaměřena na řízení výrobních procesů a monitorování a sběr dat v reálném čase, a to lokálně nebo vzdáleně. Umožňuje přímou interakci se zařízeními, ze kterých se systém skládá, a uchovává záznamy o různých druzích událostí. SCADA systémy slouží industriálním organizacím k udržení efektivity procesů, k chytrému zpracovávání dat, k řízení systému a k rychlé komunikaci systémových chyb, aby došlo k redukci času, během kterého jsou chybné procesy mimo provoz. [2]

[3] HMI – Human-Machine Interface

[4] Česky interakce člověka s počítačem.

[5] SCADA – Supervisory Control And Data Acquisition

2.2.1 Struktura SCADA systémů

Mezi základní prvky SCADA systémů patří HMI, Historian, datapoint, PLC[6] a RTU[7], které jsou níže blíže popsané.

HMI Podobně jako u DCS slouží HMI k usnadnění interakce člověka s koncovými zařízeními systému.

Historian Tento modul je zodpovědný za uchovávání a logování všech dat, která SCADA systém sbírá. Uchovaná data je pak možno zpětně analyzovat. Díky tomuto modulu je možné sledovat dění v systému během specifických časových úseků. [7]

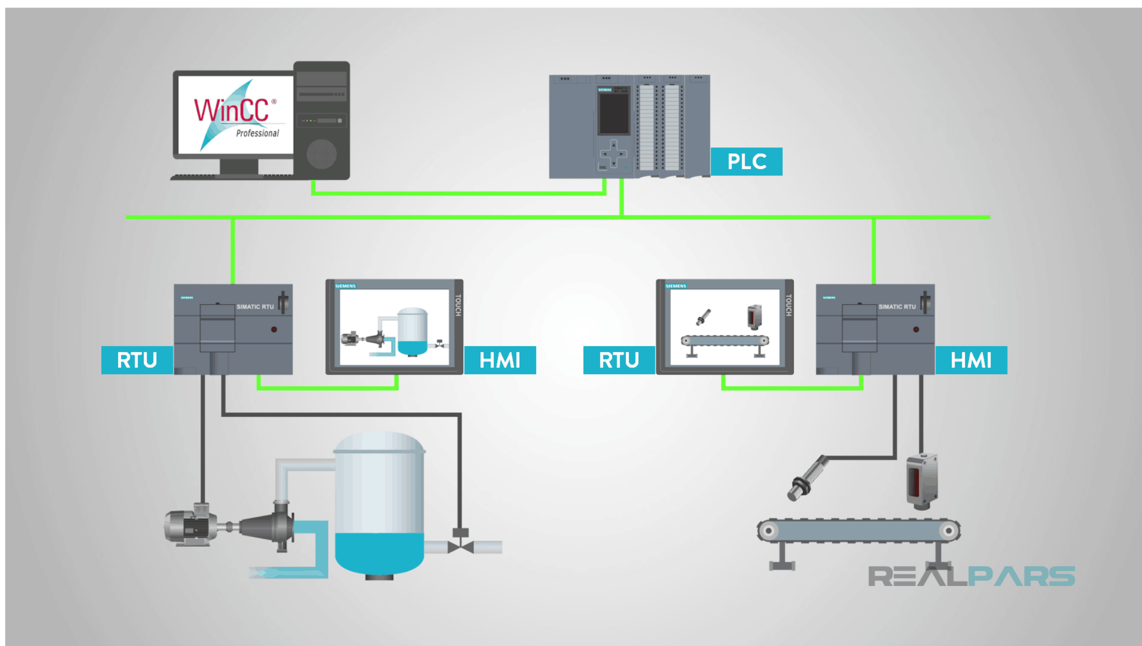
Datapoint Česky datový bod, definuje datový objekt v rámci systémové databáze. Jedná se ve své podstatě o pojmenovanou veličinu, která reprezentuje jednu vstupní, respektive výstupní hodnotu, zaznamenávanou systémem. SCADA systémy také umožňují modifikovat hodnoty datových bodů, které nikdy nejsou ovlivněny senzory či PLC zařízeními, v tom případě se používá pojem `setpoint`.[8]

PLC a RTU Jedná se o mikropočítače, které jsou schopné komunikovat s periferními zařízeními jako jsou senzory či aktivátory, a také s operačními stanicemi (HMI). Získaná data z připojených zařízení přesměrovávají do počítače, na kterém se nachází SCADA software, který je následně zpracovává a analyzuje a na jejich základě činí rozhodnutí ovlivňující chod systému.

2.2.2 Komunikační systém

Ve SCADA systémech jsou propojená PLC, RTU s periferními zařízeními a s řídicím počítačem prostřednictvím různých protokolů, které mohou být proprietární nebo standardní. PLC a RTU běží autonomně podle posledního příkazu, který přijaly od řídicího počítače. Ztráta připojení s okolními zařízeními neznamená nutně ukončení běhu zařízení. To může běžet nezávisle na ostatních zařízení a po obnově připojení je možné jej opět řídit a monitorovat. Kritické části systému mohou být redundantně propojené, aby systém mohl komunikovat i v případě výpadku jednoho komunikačního kanálu. Ve SCADA systémech se často kombinuje přímé a vzdálené propojení jednotlivých částí. Pro vzdálené propojení se používá pojem telemetrie. Mezi standardní komunikační protokoly používané ve SCADA systémech patří IEC 60870-5-101 nebo 104, IEC 61850 a DNP3. Ty jsou podporované většinou poskytovatelů SCADA systémů. Některé protokoly už obsahují rozšíření, která jim umožňují komunikovat prostřednictvím TCP/IP protokolu.

[6] PLC – Programmable Logic Controller
[7] RTU – Remote Terminal Unit
[8] https://www.dpstele.com/scada/point.php



Obrázek 2.1: Struktura SCADA systémů. Převzato z [https://realpars.com/scada/].

2.3 PLC

PLC – Programmable Logic Controller, česky programovatelný logický automat, je industriální digitální počítač, který slouží k řízení výrobních procesů. Je přizpůsobený pro napojení periferních zařízení, ze kterých se skládá technologický proces. Jedná se o jeden z prvků DCS a SCADA systémů.

2.3.1 Programování

Jednotky PLC jsou programovatelné pomocí programovacích jazyků, jež jsou definované normou IEC 61131-3. Podle této normy je možné PLC programovat pomocí:

- Function block diagram (FBD)
- Ladder diagram (LD)
- Structured text (ST)
- Sequential function chart (SFC)
- Instruction list (IL)

Tyto jazyky umožňují definovat systémy graficky či textově, popřípadě sekvenčně či paralelně.

2.3.2 Diagram funkčních bloků

Diagram funkčních bloků, anglicky Function block diagram (dále jen FBD), jak již bylo výše uvedeno jedná se o jeden z oficiálních programovacích jazyků využívaných pro programování PLC. Umožňuje programovat graficky. Je definován a popsán popsán normou IEC 61131-3.

FBD umožňuje zapouzdřovat textově definované funkce do grafických funkčních bloků.

Funkční bloky

Funkční bloky (dále jen FB) jsou základními stavebními bloky pro strukturování PLC programů. Dle normy IEC 61131-3 jsou součástí POU[9], což jsou bloky, ze kterých se skládají programy a projekty. Funkčními bloky se také zabývá norma IEC 61499, která popisuje standardy pro distribuované PLC systémy. Obě normy nahlíží na funkční bloky mírně odlišným způsobem. V této práci se inspiruji funkčními bloky podle obou norem.

Zatímco norma IEC 61131-3 definuje FB jako blok se vstupními a výstupními proměnnými, norma IEC 61499 pohlíží na FB jako na blok skládající se ze dvou částí: řízení provádění[10] a algoritmus se vstupními, vnitřními a výstupními daty. Hlavní rozdíl tedy spočívá ve způsobu, podle kterého se funkční bloky provádí.

Obě normy pohlíží na FB jako na blok se vstupními/výstupními proměnnými a vnitřní logikou, která na základě vstupu generuje určitý výstup. FB jsou vzájemně propojené přes výstupy a vstupy, což jim umožňuje předávání dat. Vzájemné propojení pak definuje výsledné chování programu. [6, 9]

Dle standardu IEC 61499 se funkční bloky dělí na:

- základní (atomické) FB,
- kompozitní FB.

O základním bloku se dá říci, že je nedělitelný. Plní určitou funkci, kterou nemá smysl z hlediska logiky celého programu rozdělit na více funkcí. Kompozitní bloky vznikly za účelem objektově orientované reprezentace programu. Jedná se o několik základních FB zkombinovaných do jednoho nového bloku, který navenek působí jako klasický FB.

Standardní funkční bloky

Norma IEC 61131-3 definuje řadu standardních FB, které pokrývají důležité oblasti funkcí PLC zařízení. Podle normy se standardní bloky dělí do pěti skupin. Následuje výpis těchto skupin a výpis několika funkčních bloků, které do nich patří. [6]

- Bistable elements - RS (Reset/Set), SR (Set/Reset)
- Edge detection - R_TRIG (Raising edge detection), F_TRIG (Falling edge detection)
- Counters - CTU (Up Counter), CTD (Down Counter), CTUD (Up Down Counter)

[9] POU – Program Organization Unit
[10] Anglicky Execution control

- Timers - TP (Pulse Timer), TON (On Delay Timer), TOF (Off Delay Timer), RTC (Real-Time Clock)
- Communication[11]

2.3.3 Strukturovaný text

Strukturovaný text, anglicky Structured Text, zkratka ST nebo STX, je jeden ze dvou textových programovacích jazyků definovaný normou IEC 61131-3. Jedná se o vysokoúrovňový jazyk strukturovaný do bloků. S ostatními jazyky normy sdílí společné komponenty[12]. Prostřednictvím tohoto jazyka je možné definovat funkce, funkční bloky a programy pro PLC zařízení. [6]

Strukturování programů

Norma IEC 61131-3 nabádá k důslednému strukturování programů pro lepší přehlednost, snadnější údržbu a jednoznačnější pohled na řešený problém a jeho řešení. Strukturování je realizováno dělením problémů na menší podproblémy a ty jsou opět děleny, dokud je to možné. Norma hovoří o dvou způsobech, které vedou ke strukturování programů: modularita a dekompozice. Článek Structuring Program Development with IEC 61131-3 [8] uvádí 7 kroků, jak dosáhnout úspěšného strukturování pomocí modularity a dekompozice. Z nich vytáhnu jen část související s programováním:

- Analýza řešení problému řízení rozložená do logických částí.
- Definice potřebných POU (Program Organisational Unit) – funkční bloky.
- Konfigurace systému definováním zdrojů, propojováním programu s fyzickými vstupy a výstupy a realizováním programů a funkčních bloků.

Příklad programu definovaného pomocí Structured Text znázorňuje výpis 2.1.

```
1  PROGRAM SampleProgram
2    VAR
3      _nodes : STRING := '["rpi1"]';
4      rpi1 : STRING := '["LuxLimit", "Pin", "GPIOOut0", "LTFB0", "bhl1750"]';
5      LuxLimit : STRING := '100';
6      Pin : STRING := '8';
7      GPIOOut0 : GPIOOut;
8      LTFB0 : LTFB;
9      bhl1750 : Compositebhl1750FB;
10   END_VAR
11
12   bhl1750();
13   LTFB0(in1 := bhl1750.out, in2 := LuxLimit);
14   GPIOOut0(a~:= LTFB0.out, pin := Pin);
15 END_PROGRAM
```

Výpis 2.1: Ukázka definice programu SampleProgram v jazyce ST.

[11]Více o těchto blocích se můžete dočíst na https://www.plcacademy.com/function-block-diagram-programming//#bit-logic-function-blocks

[12]Anglicky Common Elements

2.3.4 Rozdíl mezi DCS a SCADA

DCS i SCADA systémy sdílí spoustu společných prvků a není možné na ně nahlížet jako na dvě odlišné věci. Obě dvě technologie se zabývají monitorováním a řízením procesů, přesto se však liší ve svém zaměření. Zatímco DCS je zaměřeno na procesy, jejichž stav řídí systém, SCADA se zabývá více prací s daty a systém je řízen na základě událostí, které jsou za jeho běhu zaznamenány. Pro DCS je typické, že se skládá z komponent od jednoho výrobce, což může vést ke snadnější instalaci a ke větší integritě systému, zatímco SCADA systémy se skládají spíše z komponent od více různých výrobců a umožňují větší flexibilitu systému.

2.4 Komunikační systém

Základní komunikační systém v mém projektu má být postaven na protokolu MQTT, kterému se bude věnovat následující sekce.

2.4.1 MQTT

Message Queuing Telemetry Transport, dále jen MQTT, je nástroj určený pro přenos dat mezi chytrým zařízením (IoT[13]) a serverem. Jedná se o komunikační protokol založený na principu publikace a odběru zpráv s určitým předmětem. Je velice nenáročný, co se týče šířky pásma, jež zabírá při přenosu dat, tudíž je velice vhodný pro propojení chytrých zařízení, které obvykle nevyžadují široké přenosové pásmo. [10]

2.4.2 Vlastnosti MQTT

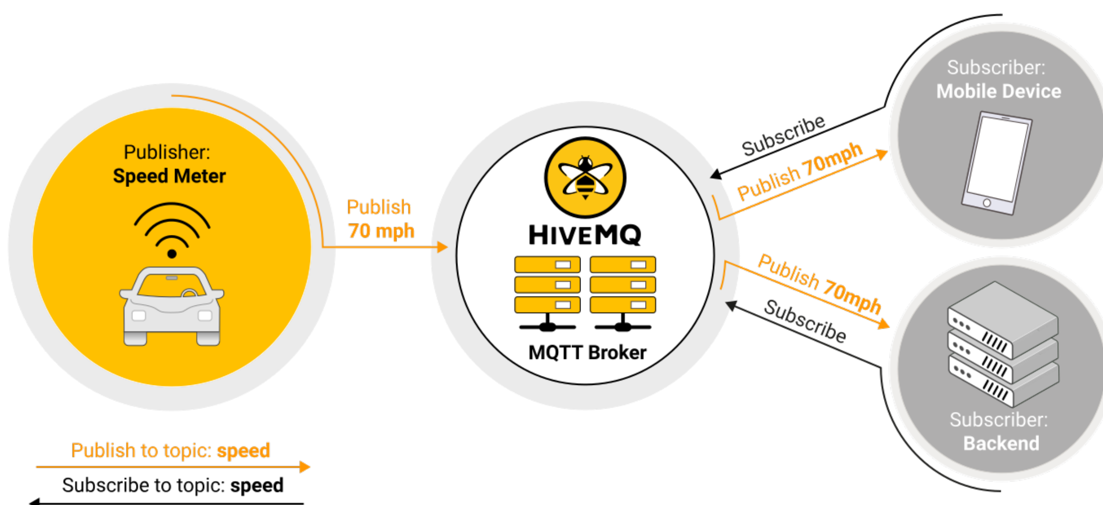
Mezi vyhledávané vlastnosti MQTT protokolu patří především jeho nenáročnost, efektivita v oblasti využití zdrojů potřebných na klientské straně a malá šířka přenosového pásma. Další vlastností je možnost oboustranné komunikace mezi připojenými zařízeními a servery, MQTT je protokol řízený událostmi. MQTT dále nabízí možnost broadcastové komunikace mezi různými skupinami zařízení, které jsou identifikovány pomocí předmětů. Protokol nabízí možnost specifikace QoS[14], která se používá jako ukazatel kvality spojení. U přenosu dat je podporováno šifrování TLS[15] a také ověřovací protokoly. Bližší informace o všech vlastnostech jsou k dispozici na [1].

2.4.3 Princip MQTT

Účastnící komunikace využívající MQTT protokol jsou následující:

Klient Koncové zařízení nebo program (většinou chytré zařízení IoT), který umožňuje otevřít internetové spojení se serverem. Klient publikuje zprávy s daným předmětem a přijímá zprávy, pro jejichž předmět má nastavený odběr. Zprávy s určitým předmětem může začít odebírat, případně jejich odběr zrušit. Klient také připojení se serverem uzavírá.

[13] IoT – Internet of Things
[14] QoS – Quality of Service, více viz https://en.wikipedia.org/wiki/Quality_of_service
[15] TLS – Transport Layer Security, více viz https://en.wikipedia.org/wiki/Transport_Layer_Security



Obrázek 2.2: Příklad MQTT komunikace. Převzato z [https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe/].

Server/Broker Prostředník mezi klienty. Jedná se opět o program či zařízení, které umožňuje posílání zpráv klientům na základě předmětů, které odebírají. Broker přijímá a ukončuje spojení s klientem a také zpracovává klientské požadavky na odběr, respektive zrušení odběru zpráv. Pokud klient zrovna není připojený, broker si ukládá zprávy, které nešly rozeslat, což je výhodné hlavně v situacích, kdy je nespolehlivé připojení. [1]

Ukázka MQTT komunikace je znázorněna na obrázku 2.2.

2.4.4 Implementace MQTT

Protokol MQTT existuje volně v mnoha implementacích pro různé programovací jazyky. Pro účely této práce se zabývám implementacemi MQTT klientů v jazyce Python. Mezi nejčastěji používané implementace patří `paho-mqtt`[16], `HBMQTT`[17] a `gmqtt`[18]. Každý z těchto klientů má své výhody i nevýhody. `Paho-mqtt` je z nich nejvíce dokumentované a má velkou podporu uživatelů a použití této implementace je nejjednodušší, proto jsem se rozhodl ji využít ve své práci.

[16] https://github.com/eclipse/paho.mqtt.python
[17] https://github.com/beerfactory/hbmqtt
[18] https://github.com/wialon/gmqtt

Kapitola 3

Použitý hardware a software

Tato kapitola se věnuje hardwarovým platformám, které jsou vhodné pro použití v centrálních distribuovaných systémech jako koncové uzly. Jsou zde popsány jejich základní vlastnosti a funkcionalita, kterou nabízí. Dále se v této kapitole nachází popis softwaru, ve kterém je možné vyvíjet programy a funkce pro výše zmíněné systémy.

3.1 Raspberry Pi

Pojem Raspberry Pi v sobě zahrnuje celou řadu malých jednodeskových počítačů. Tyto počítače jsou vyvíjeny nadací Raspberry Pi Foundation, jejíž cílem je poskytnout produkt s dostatečným výkonem za cenu, která má být přijatelná pro co největší možné spektrum zákazníků. Nadace dále podporuje rozvoj studia informačních technologií, podle toho přizpůsobují parametry počítačů a dodávaný software. [11]

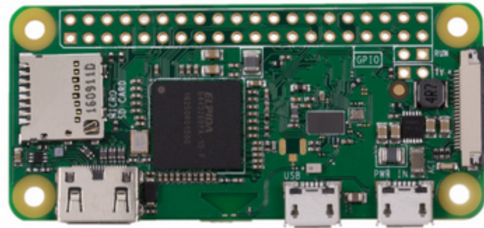
Od svého vzniku vydala nadace již několik generací počítačů, kde každá z nich existuje ve více variantách. Obsahují procesory kompatibilní s ARM instrukční sadou. Jednotlivé generace se liší jak výkonem a velikostí operační paměti, tak jejich velikostí. Největší modely mají rozměr 85.6 mm $\times$ 56.5 mm, zatímco nejmenší má 21mm $\times$ 51 mm.

Raspberry Pi Zero

Jedná se o model (obrázek 3.1), který byl poprvé představen v roce 2015. Je to jeden z menších modelů Raspberry Pi, jeho rozměry jsou 65 mm $\times$ 30 mm. Model obsahuje 1 GHz jednojádrový procesor, 512 MB operační paměti RAM, mini HDMI port pro připojení externího monitoru a micro USB OTG port. Ve své základní variantě je bez modulů Bluetooth a WiFi a bez GPIO header. Jeho rozšířená varianta Raspberry Pi Zero WH už výše zmíněné nedostatky neobsahuje. Cenově se model ve variantě WH pohybuje okolo 440 korun českých[1]. [12]

Svými malými rozměry, poměrem výkonu k ceně a rozhraním, které umožňuje připojit k němu externí periférie prostřednictvím vestavěných portů a GPIO pinů, je tento počítač vhodný pro použití v centrálních distribuovaných systémech.

[1] Cena k datu vydání této práce podle e-shopu dostupného na `https://rpishop.cz/raspberry-pi/685-raspberry-pi-zero-wh-4250236816296.html`.



Obrázek 3.1: Raspberry Pi Zero. Převzato z [https://rpishop.cz/raspberry-pi/647-raspberry-pi-zero-w-4053199547425.html].

GPIO Raspberry Pi Zero má v sobě čip BCM2835 od společnosti Broadcom [13]. Tento čip umožňuje použití následujících periférií:

- Timers
- Interrupt controller
- GPIO
- USB
- PCM / I2S
- DMA controller
- I2C master
- I2C / SPI slave
- SPI0, SPI1, SPI2
- PWM
- UART0, UART1

Raspberry Pi modely mají 40 pinovou hlavici univerzálních vstupně/výstupních pinů. Piny jsou seřazeny do dvou řad po 20. Hlavice obsahuje 12 nekonfigurovatelných pinů, kde 4 z nich poskytují stálé napětí 5 V a 3.3 V (každé napětí má dva piny) a zbylých 8 jsou s nulovým napětím – GND (ground). Ostatní piny můžou sloužit buď jako vstupní nebo výstupní. Výstupní piny můžou být nastaveny na vysokou nebo nízkou hladinu, které představují 3.3 V respektive 0 V. Vstupní piny čtou stejně jako výstupní napětí 3.3 V nebo 0 V, tedy vysoké respektive nízké napětí. Čtení umožňují pull-up a pull-down rezistory, které je možné softwarově nakonfigurovat[2]. Tabulka 3.1, zobrazuje všech 40 pinů s popisem jejich funkce.

V různých kombinacích můžou tyto GPIO piny sloužit dalším funkcím, jako je PWM (pulse-width modulation), SPI, I2C, Serial.

[2]Kromě pinů GPIO2 a GPIO3, které mají napevno pull-up rezistory.

Funkce	Číslo pinu		Funkce
3V3	(1)	(2)	5V
GPIO2	(3)	(4)	5V
GPIO3	(5)	(6)	GND
GPIO4	(7)	(8)	GPIO14
GND	(9)	(10)	GPIO15
GPIO17	(11)	(12)	GPIO18
GPIO27	(13)	(14)	GND
GPIO22	(15)	(16)	GPIO23
3V3	(17)	(18)	GPIO24
GPIO10	(19)	(20)	GND
GPIO9	(21)	(22)	GPIO25
GPIO11	(23)	(24)	GPIO8
GND	(25)	(26)	GPIO7
GPIO0	(27)	(28)	GPIO1
GPIO5	(29)	(30)	GND
GPIO6	(31)	(32)	GPIO12
GPIO13	(33)	(34)	GND
GPIO19	(35)	(36)	GPIO16
GPIO26	(37)	(38)	GPIO20
GND	(39)	(40)	GPIO21

Tabulka 3.1: Část výpisu příkazu `pinout` na zařízení Raspberry Pi Zero popisující GPIO hlavici.

3.2 Software

Softwarové možnosti pro vývoj distribuovaných řídících systémů na platformě Raspberry Pi Zero se značně odvíjejí od operačního systému, na kterém zařízení běží. RPi podporuje spoustu nenáročných operačních systémů, které jsou uzpůsobené různým potřebám uživatelů. V tomto projektu budu pracovat s nejpoužívanějším z nich, Raspberry Pi OS, který je oficiálním operačním systémem. RPi OS je založený na Debianu a je optimalizovaný pro hardware Raspberry Pi modelů. Obsahuje velké předinstalovaných balíčků programů. Dostupnost softwaru pro tento operační systém je tedy velmi otevřená. Pro vývoj DCS je možné využít téměř libovolného programovacího jazyka, který je běžně podporovaný Linuxovým systémem. Tento projekt bude napsán v jazyce Python verze 3, jelikož je v RPi OS už předinstalován a podporuje programování GPIO modulu.

Kapitola 4

Analýza a návrh systému

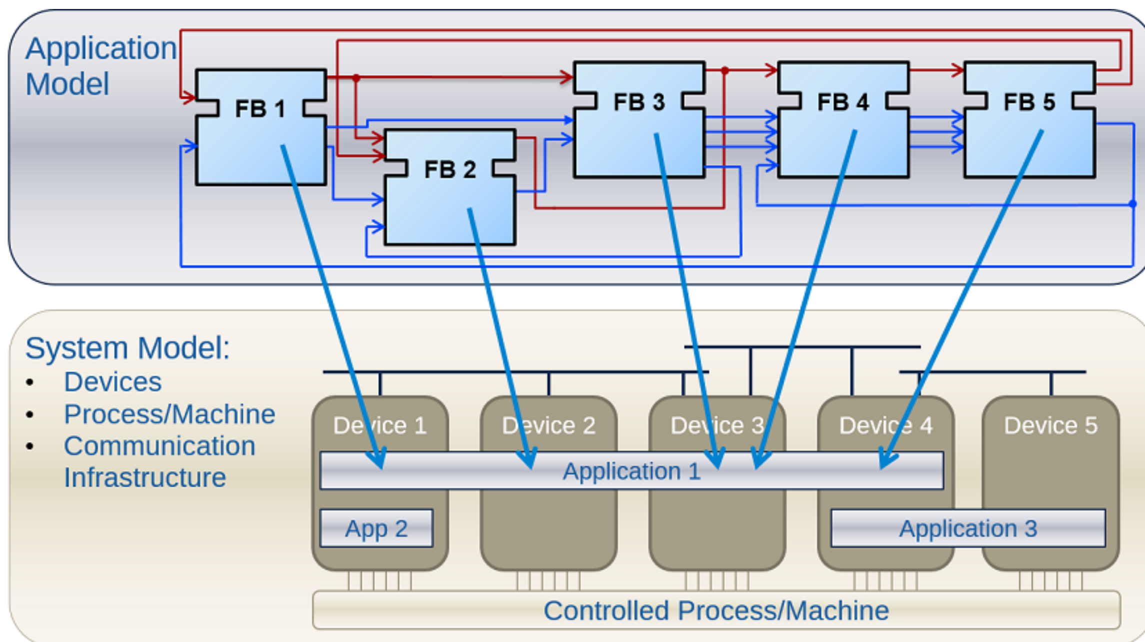
Obsahem této kapitoly je analýza prostředků umožňující vytvoření vlastního distribuovaného řídicího systému s dynamicky modifikovatelnými uzly komunikujícího přes protokol MQTT, dále pak návrh tohoto systému. Důraz je v této kapitole kladen na operační systém uzlů a HMI systému.

4.1 Distribuovaný řídicí systém

Na DCS lze nahlížet z více různých úhlů. V prvé řadě je zde systémový model. Ten tvoří jednotlivé uzly systému, řízené procesy či zařízení a komunikační infrastruktura. Další součástí systému je pak model aplikační, který definuje jeho chování. Ten se skládá z funkčních bloků a jejich vzájemného propojení. Aplikace nemusí běžet pouze na jednom zařízení (uzlu systému), naopak může být rozdělena a dílčí funkční bloky rozmístěny mezi více uzly. Norma IEC 61499 dokonce umožňuje, aby takových aplikací v rámci jednoho systému mohlo běžet i více. Obrázek [4.1] demonstruje oba výše uvedené modely a také způsob, jakým je aplikace rozdělená na více uzlech. Systém, který je v rámci této práce vyvíjen, využívá výše uvedených modelů, s tím rozdílem, že systém primárně podporuje běh pouze jedné aplikace. Ta se však může skládat z více menších aplikací, kde každá z nich může řešit jiný problém. [9]

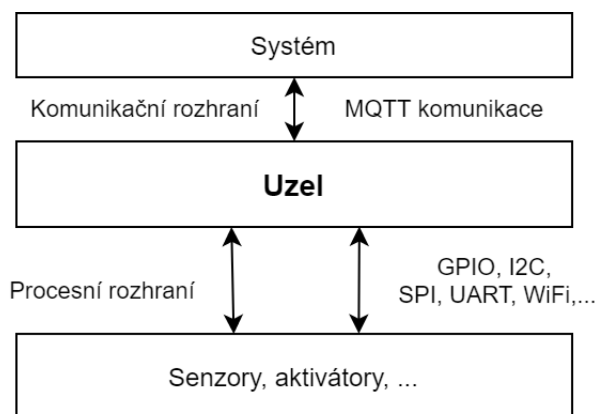
4.2 Uzly DCS

Jako uzly distribuovaného řídícího systému slouží počítače Raspberry Pi Zero, které běží na operačním systému Raspberry Pi OS, který je založený na Debian GNU/Linux. Uzel je základní jednotkou distribuovaného řídícího systému. Základní úlohou uzlu je zpracovat vstupní data, vygenerovat příslušný výstup a ten dále distribuovat dalším uzlům. U uzlu se rozlišují dva typy rozhraní, pomocí kterých komunikuje se svým okolím, a sice rozhraní komunikační a procesní. Pomocí komunikačního rozhraní uzel komunikuje s ostatními uzly v rámci dané sítě. Pro tuto komunikaci se využívá výhradně protokol MQTT. Procesní rozhraní pak slouží ke komunikaci s připojenými periferními zařízeními, jako jsou senzory či aktivátory. S těmi pak komunikuje prostřednictvím rozhraní, jež poskytuje hardware a software zařízení, které slouží jako uzel (v případě této práce zařízení Raspberry Pi



Obrázek 4.1: Systémový a aplikační model distribuovaných řídících systémů. Převzato z [https://www.eclipse.org/4diac/en_help.php?helppage=html/before4DIAC/iec61499.html].

Zero). Rozhraní uzlu jsou znázorněna na obrázku 4.2. Uzel sám o sobě není závislý na ostatních uzlech v síti, ale na jejich výstupech, to znamená, že pokud je uzel z libovolného důvodu mimo provoz, neovlivní tím provoz ostatních uzlů, což je jedna z hlavních vlastností distribuovaných systémů.



Obrázek 4.2: Komunikační a procesní rozhraní uzlu DCS.

4.3 Uživatelská aplikace

Uživatelskou aplikací se má na mysli úloha, kterou řeší daný distribuovaný řídící systém. Její vlastností je rozmístění svých částí na jednotlivé uzly systému. Tyto části tvoří funkční

jednotky, jejichž běh má smysl na daném uzlu. Tyto funkční jednotky se nazývají funkční blok.

Aplikace je tedy definována jednotlivými instancemi funkčních bloků a jejich vzájemným propojením. Kromě funkčních bloků aplikaci tvoří ještě speciální proměnné, které umožňují její parametrizaci. Jedná se v podstatě o proměnné, které mohou sloužit jako konstantní vstup funkčního bloku, případně do nich může být ukládán výstup funkčního bloku.

Podstatnou vlastností aplikace je možnost hierarchického skládání funkčních bloků. To je umožněno díky kompozitním funkčním blokům, které jsou blíže popsány v sekci [4.4].

4.4 Funkční blok

Funkční bloky použité v této práci jsou inspirované především normami IEC 61131-3, která se zabývá programovatelnými logickými automaty (PLC), a IEC 61499, jež je přímo zaměřená na funkční bloky v řídicích systémech. Funkční blok je definován svými vstupy, výstupy a funkcí, která na základě vstupů generuje výstupy. Funkční blok může také obsahovat pomocné lokální proměnné. Bloky jsou vzájemně propojené skrze své vstupní a výstupní porty, a toto jejich propojení tvoří uživatelskou aplikaci. Samotný blok však neřeší, na co je napojený, respektive co je napojené na něj. Přenos dat z výstupního portu je zodpovědností runtime platformy.

4.4.1 Dělení funkčních bloků

V tomto projektu rozlišuji funkční bloky podle složitosti a funkce.

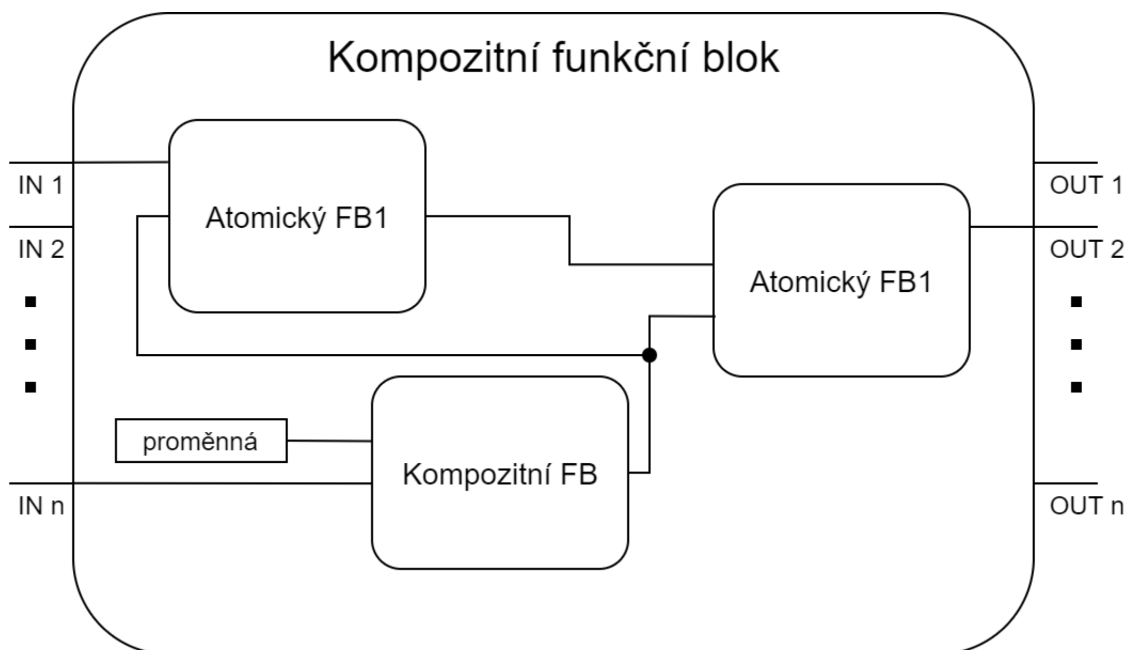
Dělení podle složitosti

Atomické FB Základní nedělitelný blok, který má jednoduchou funkci. Touto funkcí se nemyslí pouze známé aritmetické či logické operace, ale obecně libovolný algoritmus zpracovávající vstupy a generující výstupy. Funkční blok tedy může implementovat i sekvenční logiku, zpoždění, automat atd.

Kompozitní FB Bloky složené z jednoho a více atomických nebo kompozitních bloků, které se navenek tváří jako atomické funkční bloky. Umožňují tvořit hierarchii funkčních bloků, což vede k snadnější definici komplexnějších uživatelských aplikací. Na rozdíl od atomických funkčních bloků musí řešit propojení bloků, ze kterých se skládá. Struktura kompozitního funkčního bloku je znázorněna v obrázku 4.3. Kompozitní blok, tak jak je používán v mém systému, je chápán jako generický blok, tedy chová se jako aplikace. Tak jej definuje norma IEC 61499. Je však možné připustit i další pohled na tento blok, o čemž je napsáno více v příloze [B].

Dělení podle funkce

Servisní FB Úlohou těchto bloků je řešit problémy, které nejsou přímo spojené s uživatelskou aplikací jako takovou. Zabývají se především správou samotné aplikace a uzlu,



Obrázek 4.3: Návrh struktury kompozitního funkčního bloku.

na kterém běží. Tyto funkční bloky jsou součástí runtime platformy uzlů a úzce souvisí se servisní komunikací systému, která je blíže popsána v sekci [4.5]. Konkrétní servisní bloky jsou popsány v sekci [5.2.4].

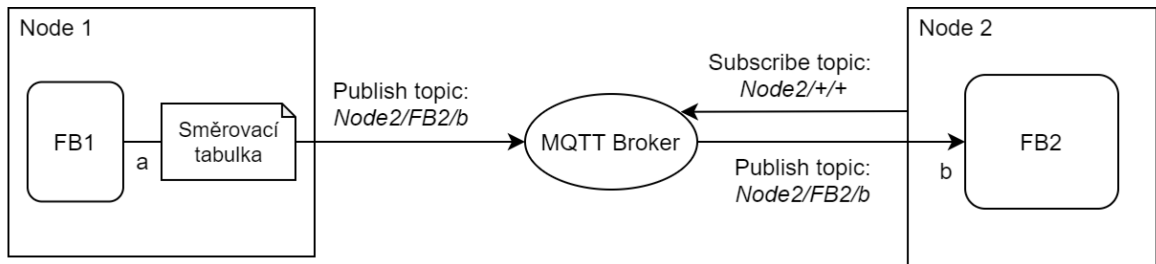
Uživatelské Uživatelské funkční bloky souvisí přímo s funkcionalitou uživatelské aplikace. Jejich funkce je definovaná uživatelem. Jedná se o bloky atomické i kompozitní.

4.5 Komunikace DCS

Komunikaci v distribuovaných systémech tvoří především přenos dat mezi jednotlivými uzly. Je realizována prostřednictvím technologie MQTT. Základní komunikační strukturu tedy tvoří broker a k němu připojení klienti. Základní úlohou brokeru je směrovat přijaté zprávy na správné uzly, což je mu umožněno pomocí předmětů (topic) jednotlivých MQTT zpráv. Zprávy přenášené mezi uzly se dělí na aplikační a servisní. Aplikačními zprávami se má na mysli přenos dat související s funkcionalitou daného systému. Pro účely správy uzlů slouží zprávy servisní, které umožňují nastavovat a měnit funkci uzlů, čímž se realizuje jejich modifikovatelnost v rámci daného systému. Servisní zprávy dále slouží k monitorování stavu uzlů, respektive celého systému.

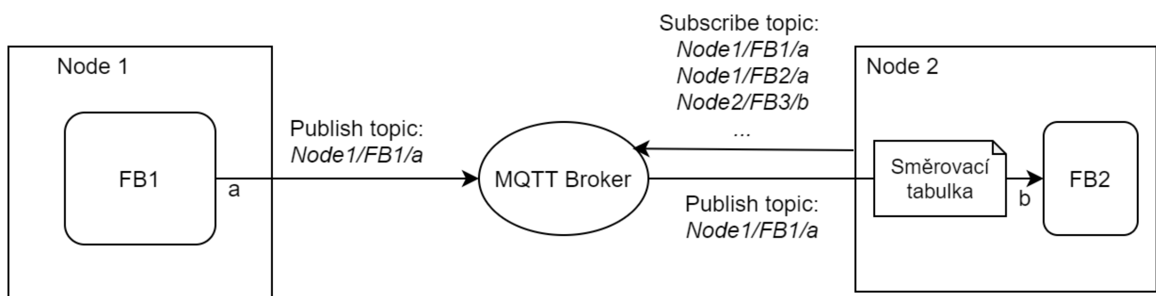
Z pohledu uživatelské aplikace se jedná o komunikaci mezi jednotlivými instancemi funkčními bloky, ze kterých se skládá. Zde se nabízí více možností, jak zprávy v systému adresovat. První možností je, že FB publikuje své výstupy pod předmětem skládajícím se z názvu instance toho bloku a jeho daného výstupního portu. FB, jejichž vstup je výstupem z daného bloku, by pak odebíraly zprávy s výše uvedeným předmětem. U této varianty je směrování řešeno na úrovni cílového uzlu, jehož povinností je zařídit odběr zpráv pro každý port

každého funkčního bloku, jehož výstup je vstupem nějakého funkčního bloku na daném uzlu. Tuto možnost demonstruje obrázek 4.4.



Obrázek 4.4: Směrování zpráv, které probíhá před publikováním zprávy.

Druhá možnost je, že funkční bloky publikují zprávy s předmětem tvořeným z názvu instance cílového FB a jejich vstupních portů. Směrování probíhá na zdrojovém uzlu, který na základě směrovací tabulky publikuje zprávy s předmětem příslušného cílového funkčního bloku. Povinností cílového uzlu je pouze odběr zpráv s předmětem `node_name/+/+`, kde `+` představuje libovolnou hodnotu. Tuto možnost demonstruje obrázek 4.5.



Obrázek 4.5: Směrování zpráv, které probíhá až po přijmutí zprávy na straně cílového uzlu.

a třetí možnost řeší směrování jednou globální směrovací tabulkou, která běží na samostatném uzlu, jenž slouží jako hlavní směrovač celé aplikace. Tuto možnost jsem se rozhodl nerealizovat, protože by vyžadovala vytvoření dalšího speciálního programu, který by sloužil pouze k účelům směrování. Jelikož existují možnosti, jak řešit směrování v rámci operačního systému uzlů, rozhodl jsem se zvažovat první dvě možnosti.

První dvě možnosti jsou prakticky rovnocenné, ale liší se způsobem implementace směrování. Je potřeba vzít v úvahu způsob vykonávání funkčních bloků. Protože je výkon FB řízen jeho vstupními daty, je vhodné, že v příchozí zprávě je přímo uvedena instance, která má běžet. To by u první varianty bylo značně komplikovanější, jelikož by se informace o cílové instanci musela ještě vybrat podle směrovací tabulky. Výhodou druhé možnosti je automatické zpracovávání příchozích zpráv na cílovém uzlu, které umožňuje hierarchická struktura předmětu. Pro cílový uzel systému je pak jednodušší mapovat vstupní data na port funkčního bloku.

V rámci optimalizace směrování je také vhodné uvažovat nad komunikací funkčních bloků v rámci jednoho uzlu. Bez jakýchkoliv optimalizací by totiž veškerá uživatelská data putovala vždy přes broker, což zabere značně více času, než kdyby si bloky předaly data lokálně v rámci daného uzlu. Úskalím této optimalizace je ztráta možnosti vzdáleného monitorování

toku dat v aplikaci, které je možné díky publikaci a odběru zpráv. Tento problém je řešitelný použitím speciálního monitorovacího funkčního bloku, jehož vstupy by obsahovaly výstupy z bloků, které je třeba monitorovat.

Návrh formátu zpráv

Zprávy se v tomto projektu dělí na servisní a aplikační. Každá z nich má mírně odlišný formát přizpůsobený jejich funkcionalitě. Obecně je MQTT zpráva tvořena z předmětu a přenášené informace. Předmět zprávy slouží k jejímu směřování a je odlišný pro zprávy servisní a uživatelské. Pro publikování MQTT zpráv je možné použít libovolný software, který toto umožňuje. Na linuxovém systému je jednou z možností využít program `mosquitto_pub`, který je součástí balíčku `mosquitto-clients`. Publikace zprávy tímto programem se pak volá příkazem:

```
mosquitto_pub -h hostname -m message -t topic
```

kde `hostname` je IP adresa MQTT brokeru, `message` je obsah publikované zprávy a `topic` je předmět zprávy.

Servisní zprávy

Jak bylo již výše zmíněno, servisní zprávy se týkají správy uzlů daného systému. Správa uzlu v sobě zahrnuje problémy typu: instalace/odinstalace funkčních bloků, jejich aktivace a deaktivace, nastavení směrování zpráv mezi jednotlivými bloky uživatelské aplikace, dotazování na stav uzlu, popřípadě stav konkrétní instance funkčního bloku, který na daném uzlu běží. Pro směrování servisních zpráv se využívá identifikátor uzlu. Obsah zprávy je pak určen problémem, který zpráva řeší. Obecně je obsah servisní zprávy ve tvaru:

```
{Action,Payload},
```

kde `Action` odpovídá servisní akci a `Payload` obsahuje data, která jsou potřeba pro danou akci.

Následuje seznam servisních zpráv, které jsem nadefinoval pro účely tohoto projektu:

Load Instalace funkčního bloku na daný uzel systému. Data pro tuto servisní akci obsahují zdrojový kód funkčního bloku. Viz příklad 4.6.

Unload Odinstalace funkčního bloku z daného uzlu. Zpráva obsahuje název funkčního bloku, který má být z uzlu odinstalován. Viz příklad 4.7.

Activate Instanciace funkčního bloku. Zpráva musí obsahovat název třídy, název instance nového funkčního bloku a případné parametry pro vznik instance, které jsou specifické pro daný funkční blok. Viz příklad 4.8. Aktivace kompozitních bloků je o něco složitější, jelikož kompozitní blok potřebuje znát informace o svých vnitřních funkčních blocích, o jejich propojení a o případných proměnných. Tyto informace se předávají jako parametry. Jako

```
{"Load","""class FB_class1(Task):
        def __init__(self):
                pass
        def run(self):
                self.publish("out",True)"""}

{"Load","""class FB_class2(Task):
        def __init__(self):
                self.in1=""
                self.in2=""
        def run(self):
                self.publish("out",True)"""}
```

Př. 4.6: Nahrání kódu v jazyce Python tříd `FB_class1` a `FB_class2` na uzel systému.

```
{"Unload", "FB_class1"}
```

Př. 4.7: Odstranění třídy `FB_class1` z uzlu.

název třídy se použije klíčové slovo `_compositeBlock`. Příklad 4.8 demonstruje aktivaci kompozitního funkčního bloku.

```
{"Activate",{"FB_class1", "FB1", {}}}

{"Activate",{"_compositeBlock", "CFB1", {
{ {"FB_class2","FB1",{}},{"FB_class2","FB2",{}} },
{ ("variable","value") },
{ {"in1","","FB1","in1"},{"in2","","FB1","in2"},{"FB1","out","FB2","in1"},
{"variable","","FB2","in2"} } }}}
```

Př. 4.8: Vytvoření instance `FB1` třídy `FB_class1`, bez vstupních parametrů, a instance kompozitního bloku `CFB1` s vnitřními bloky `FB1` a `FB2` třídy `FB_class2`, s proměnnou `variable` s hodnotou „`value`" a s nastavením jeho vnitřního propojení.

Remove Odstranění instance daného funkčního bloku. Zpráva musí obsahovat název instance, která má být na daném uzlu odstraněna. Viz příklad 4.9.

```
{"Remove","FB1"}
```

Př. 4.9: Odstranění instance `FB1` z uzlu.

AddRoute Přidání záznamu do směrovací tabulky uzlu nebo kompozitního funkčního bloku. Zpráva musí obsahovat informace o zdrojovém a cílovém portu a instanci a o uzlu, na kterém se cílová instance nachází. Viz příklad 4.10.

RemoveRoute Odebrání záznamu ze směrovací tabulky uzlu nebo kompozitního funkčního bloku. Obsah zprávy viz `AddRoute`. Příklad 4.11.

```
{"AddRoute",
{"FB1","out","node1", "CFB1", "in1"}}

{"AddRoute",
{"variable","","node1", "CFB1", "in2"}}
```

Př. 4.10: Vytvoření záznamu do směrovací tabulky. Port `out` instance `FB1` je spojený s portem `in1` kompozitního bloku `CFB1` na uzlu `node1`.

```
{"RemoveRoute",
{"FB1","out","node1", "CFB1", "in1"}}
```

Př. 4.11: Odstranění záznamu ze směrovací tabulky.

AddVar Vytvoření proměnné na daném uzlu nebo kompozitním funkčním bloku. Obsah zprávy musí obsahovat název nově vytvářené proměnné. Viz příklad 4.12.

```
{"AddVar","variable"}
```

Př. 4.12: Vytvoření proměnné `variable`.

SetVar Nastavení hodnoty proměnné na daném uzlu nebo kompozitním funkčním bloku. Obsah zprávy musí obsahovat název proměnné a její novou hodnotu. Viz příklad 4.13.

```
{"SetVar","variable","True"}
```

Př. 4.13: Nastavení hodnoty proměnné `variable` na hodnotu `True`.

RemoveVar Odstranění proměnné na daném uzlu nebo kompozitním funkčním bloku. Obsah zprávy musí obsahovat název proměnné, která má být odstraněna. Viz příklad 4.14.

```
{"RemoveVar","variable"}
```

Př. 4.14: Odstranění proměnné `variable`.

Status Tato zpráva slouží ke sdílení informací o uzlech mezi uzly samotnými. V rámci této zprávy může jeden uzel požadovat informace o druhém uzlu, konkrétně o jeho třídách, instancích a proměnných. Zpráva v tomto případě obsahuje klíčové slovo `Get` a název zdrojového uzlu, aby cílový uzel věděl, kam má své informace zaslat. Viz příklad 4.15. Uzel, který je dotázán na své informace, posílá svá data zpět svému tazateli. V tomto případě musí zpráva obsahovat klíčové slovo `Post` a samotná data uzlu. Viz příklad 4.16.

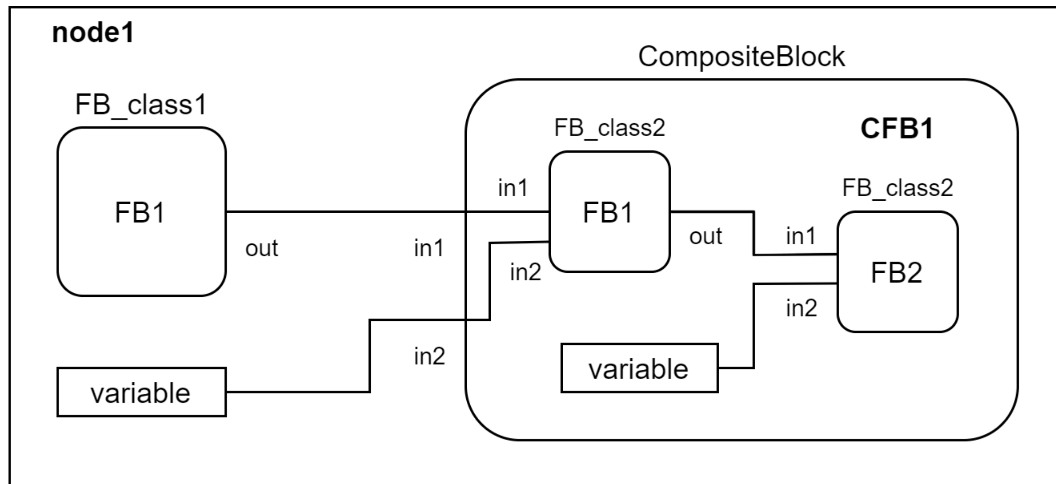
```
{"Status",{"Get","node1"}}
```

Př. 4.15: Zaslání požadavku uzlu node1 na informace o cílovém uzlu.

Publikováním zpráv z příkladů 4.6, 4.8, 4.10, 4.12 a 4.13 s předmětem zprávy `node1`, jež odpovídá identifikátoru uzlu, vznikne aplikace, kterou zobrazuje obrázek 4.17.

```
{"Status",{"Post","{'classes': ['FB_class1','FB_class2'],
'instance': ['FB1','CFB1','CFB1/FB1','CFB1/FB2'],
'variables': [('variable', 'True'),('CFB1/variable', 'True')],
'info': [('Node name','node1'),('MQTT Broker IP','192.168.0.136'),
('IP address','192.168.0.67')],
'routes': [('FB1/out', ['CFB1/in1']), ('variable', ['CFB1/in2']),
('CFB1/FB1/out', ['CFB1/FB2/in1']),('CFB1/variable', ['CFB1/FB2/in2'])]}}}
```

Př. 4.16: Zdrojový uzel posílá své informace.



Př. 4.17: Příklad aplikace nadefinované sekvencí MQTT servisních zpráv.

Podpora hierarchie funkčních bloků

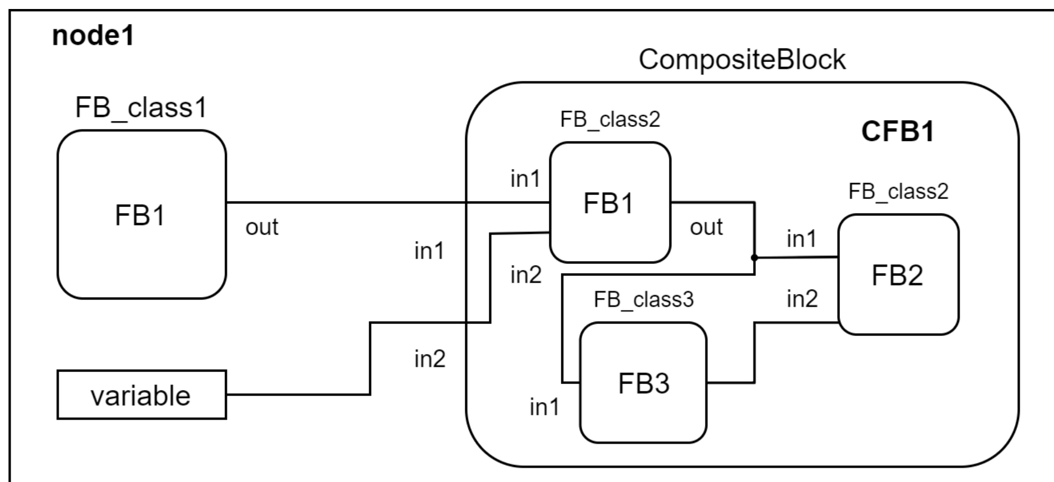
Vzhledem k možné hierarchii funkčních bloků v rámci uživatelské aplikace je možné zprávy, u kterých to dává smysl, aplikovat v libovolné úrovni zanoření funkčních bloků. V obsahu zprávy se to projeví přidáním cesty, tvořené z názvů rodičovských kompozitních bloků oddělených znakem „/“, k názvu instance či proměnné.

Aplikační zprávy

Pro zprávy posílané mezi uživatelskými funkčními bloky stačí jednodušší formát, než je u zpráv servisních, jelikož obsahem zprávy jsou vždy pouze uživatelská data. Tento velice nekomplexní formát je dostačující, jelikož informace o směrování jsou předávány v předmětu zprávy a žádná jiná data se mezi bloky neposílají. Příklad 4.18 ukazuje celou MQTT zprávu s předmětem i obsahem.

```
Topic:    node2/FB2/in1
Payload:  "True"
```

Př. 4.18: MQTT zpráva zaslána na port in1 funkčního bloku FB2 na uzlu node2. Obsahem zprávy je hodnota „True“.



Př. 4.19: Podoba aplikace z obrázku 4.17 po dynamické rekonfiguraci.

4.5.1 Dynamická rekonfigurace

Servisní zprávy, které spravují uživatelskou aplikaci, je možné publikovat, kdykoliv je systém, respektive jeho uzly, v provozu. Tato skutečnost umožňuje modifikovat části aplikace za jejího běhu. Je možné provádět veškeré operace, které jsou zrealizovatelné prostřednictvím servisních zpráv. Funkční bloky je možné dynamicky vytvářet, odstraňovat či propojovat s dalšími bloky, aniž by to ovlivnilo chod aplikace. Na obrázku 4.19 je znázorněna podoba uživatelské aplikace z obrázku 4.17 po dynamické rekonfiguraci, jež tvoří sekvence servisních zpráv ve výpise 4.1.

```
{"RemoveVar", "CFB1/variable"}
{"RemoveRoute", {"CFB1/variable","","CFB1/FB2","in2"}}

{"Load", """class FB_class1(Task):
  def __init__(self):
    self.in1=""

  def run(self):
    self.publish("out",True)"""}
{"Activate", "FB_class3","CFB1/FB3"}
{"AddRoute", {"CFB1/FB3","out","CFB1/FB2","in2"}}
{"AddRoute", {"CFB1/FB1","out","CFB1/FB3","in1"}}

{"SetVar", "variable", "False"}
```

Výpis 4.1: Sekvence MQTT servisních zpráv, jež vedou k dynamické modifikaci aplikace z obrázku 4.17.

4.6 Runtime platforma

Jedná se o aplikaci, která běží na každém uzlu distribuované sítě a obstarává jeho veškerou funkcionalitu. Mezi základní úlohy této platformy patří umožnění komunikace mezi uzly systému, respektive mezi funkčními bloky, které jsou nainstalované na ostatních uzlech,

spouštění funkčních bloků a monitorování stavu. Runtime platforma tvoří běhové prostředí pro uživatelské aplikace. Platforma je dělená do modulů, které řeší jednotlivé úlohy.

Modul komunikace

Účelem tohoto modulu je vytvořit rozhraní runtime platformy, které jí umožní propojení s ostatními platformami běžícími na ostatních uzlech systému. Komunikace je zajištěna pomocí přenosového protokolu MQTT. Pro připojení platformy do systému je třeba, aby se uzel připojil na systémový MQTT server a nastavil si odběr příslušných zpráv. Modul dále řeší přijímání a odesílání zpráv. Přijaté zprávy předává dále ke zpracování a naopak odchozí zprávy na základě informací o zdrojovém funkčním bloku a jeho portu přeposílá pod správným předmětem na broker, který ji přesměruje na cílový uzel.

Modul konfigurace

Pro nastavení samotné runtime platformy slouží modul konfigurace. Tento modul definuje uživatelskou aplikaci, respektive tu část, která běží na daném uzlu. Rozhraní tohoto modulu umožňuje správu funkčních bloků a jejich instancí, vnitřních proměnných, směrovací tabulky a dat celého programu. Modul řeší způsob uložení dat o uživatelské aplikaci na všech možných úrovních hierarchického zanoření bloků. Jelikož je každý kompozitní blok definován prakticky stejnými informacemi jako celá uživatelská aplikace, využívají se prostředky tohoto modulu i pro jejich reprezentaci.

Hlavní modul

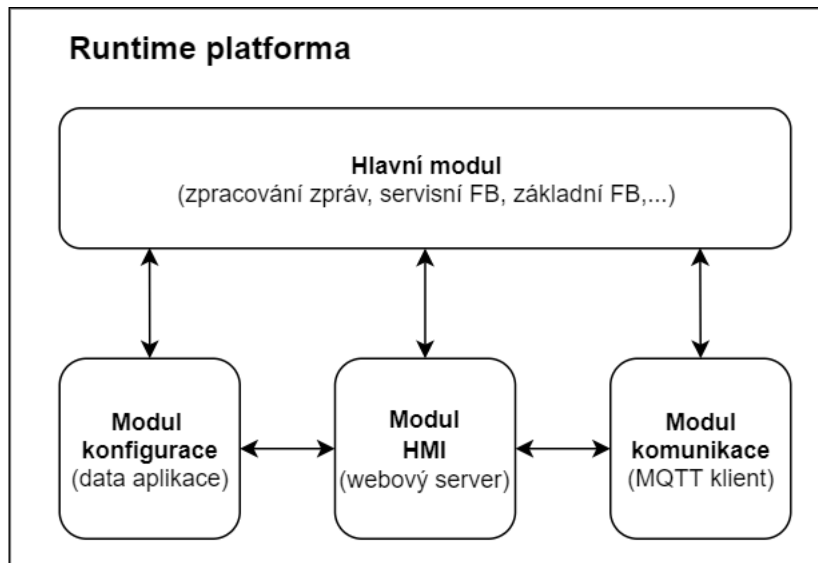
Tento modul se zabývá instalací, správou a během uživatelské aplikace. Definuje rozhraní pro uživatelské funkční bloky, díky čemuž je možné je v runtime aplikaci instalovat, modifikovat, odstraňovat a provádět.

Zpracování servisních zpráv

Pro zpracování servisních zpráv modul využívá servisních funkčních bloků, které jsou v něm přímo nadefinované. Každému typu servisní zprávy přísluší jeden servisní funkční blok, jehož úkolem je realizovat požadavek zprávy v rámci runtime platformy. Modul tedy definuje 10 bloků, jejichž názvy jsou shodné s názvy typů servisních zpráv, tak, jak jsou uvedeny v sekci 4.5. Obsah přijaté zprávy je předán jako parametr danému servisnímu bloku, který pak na jeho základě provádí svou činnost.

Zpracování uživatelských zpráv

Dále modul řeší zpracování uživatelských zpráv. Modul identifikuje cílovou instanci funkčního bloku na základě předmětu přijaté zprávy. Následně dané instanci předá obsah zprávy na správný vstupní port a volá její funkci, která implementuje algoritmus bloku.



Obrázek 4.20: Návrh struktury runtime platformy – propojení modulů.

Rozhraní uživatelského funkčního bloku

Hlavní modul definuje třídu, která slouží jako základ pro veškeré uživatelem nadefinované funkční bloky. Tato třída definuje funkci, která definuje algoritmus funkčního bloku a pomocné funkce, které umožňují publikování výstupů.

Modul HMI

Tento modul definuje servisní uživatelské rozhraní celého systému. Cílem rozhraní je umožnit vývojáři, který tvoří pro systém svou uživatelskou aplikaci, náhled na vnitřní data runtime platformy na všech uzlech systému. HMI existuje ve formě webového serveru, který běží v rámci runtime platformy. Modul komunikuje s modulem konfigurace a zobrazuje vývojáři data o funkčních blocích a jejich instancích, proměnných a směrovacích tabulkách, které se nacházejí na daných uzlech. HMI dále umožňuje modifikaci výše uvedených prvků aplikace prostřednictvím formulářů, které jsou zpracované a převedené na servisní zprávy, jež jsou zaslány příslušným uzlům. Modul také nabízí možnost sledovat data na vstupních a výstupních portech jednotlivých funkčních bloků v reálném čase, což vede ke snadnějšímu procesu ladění uživatelské aplikace.

Propojení modulů

Hlavní modul využívá pro svůj běh všechny výše uvedené moduly runtime platformy. Využívá dat o aplikaci, které jsou uloženy v rámci konfiguračního modulu, a pro komunikaci s ostatními uzly využívá modulu komunikace. V hlavním modulu vzniká instance webového rozhraní modulu HMI, které také využívá prostředků modulů komunikace a konfigurace pro svůj správný běh. Propojení modulů demonstruje obrázek 4.20.

Kapitola 5

Implementace

Tato kapitola se věnuje implementaci projektu a jeho testování na demo aplikaci. Jsou zde popsány důležité moduly, třídy a funkce, ze kterých se projekt skládá a které řeší problémy uvedené v předchozí kapitole [4]. Implementace probíhala přes vzdálené připojení výhradně na zařízeních Raspberry Pi Zero WH, jež jsou cílová koncová zařízení tohoto projektu. Projekt je implementován v jazyce Python3. Kapitola dělí projekt na dvě části – runtime platforma a inicializační skript – a postupně se věnuje oběma z nich.

5.1 Použité knihovny

Pro účely tohoto projektu jsem používal několik standardních knihoven jazyka Python. Pro větvení programu do více vláken, aby nedocházelo k blokaci běhu programu některými procesy, je použita knihovna `threading`. Připojení k MQTT brokeru realizuje knihovna `paho.mqtt`, konkrétně z ní používám modul `client`, který nabízí stejnojmennou třídu, která obstarává veškerou funkcionalitu potřebnou pro klienta MQTT komunikace. HMI, které je implementováno formou lokálního webového serveru, využívá tříd `BaseHTTPRequestHandler` a `HTTPServer` z knihovny `http.server`, která implementuje HTTP servery. Mezi další knihovny, které řeší méně podstatné záležitosti v projektu, patří: `urllib.parse`, `time`, `sys`, `collections` a `re`.

5.2 Runtime platforma

Runtime platforma tvoří nejobsáhlejší část celého projektu. Jedná se o samotný operační systém uzlu distribuovaného řídicího systému. Úkolem této aplikace je správa a řízení běhu funkčních bloků a komunikace s ostatními uzly v systému. Aplikace je členěna do vzájemně komunikujících modulů, které implementují její jednotlivé logické části.

5.2.1 Modul konfigurace

Konfigurační modul se nachází v souboru `config.py`. Jedná se o modul, který řeší veškerou problematiku spojenou s konfigurací aplikace na daném uzlu, a zároveň ukládá data o stavu aplikace. Modul obsahuje třídu `Config()`, jejíž atributy tvoří proměnné obsahující základní

informace o aplikaci, jako jsou: název (identifikátor) uzlu, IP adresa MQTT brokeru, třídy a instance funkčních bloků, proměnné a směrovací tabulka. Třída dále nabízí funkce pro nastavování či získání výše uvedených proměnných.

Reprezentace dat aplikace a uzlu

Pro reprezentaci dat uživatelské aplikace a uzlu, na kterém aplikace běží, využívá modul standardní datové struktury, jež nabízí jazyk Python. Různé prvky aplikace však vyžadují odlišný způsob reprezentace. Následuje popis uložení dat v rámci modulu konfigurace:

Základní informace o uzlu Jako základní informace uzlu se považuje jméno, neboli identifikátor uzlu, a IP adresa MQTT brokeru. Jedná se o jednoduché hodnoty, a proto jsou uložené v proměnných typu `string`.

Třídy funkčních bloků Modul konfigurace si ukládá názvy nainstalovaných tříd, které reprezentují atomické bloky v klasickém seznamu. Při aktivaci (instanciaci) funkčního bloku pak na základě tohoto seznamu dochází ke kontrole, zda je v runtime platformě třída nainstalovaná. Pro přidávání a odstraňování tříd z tohoto seznamu slouží funkce `addClass(name)` a `removeClass(name)`.

Instance funkčních bloků Pro uchování samotných instancí funkčních bloků je použita datová struktura `dictionary` (česky slovník), která se chová jako asociativní pole hodnot. Ukládá se dvojice tvořená z názvu instance a samotného objektu instance. Objekt instance je přístupný skrze funkci `getTask(name)`, která jej vyhledá a vrátí na základě hodnoty `name`. Vzhledem k hierarchické podstatě bloků funkce umožňuje rekurzivně vyhledávat i instance vnitřních bloků kompozitů, které mají svou vlastní instanci třídy `Config()`, do kterých si ukládají informace o svých datech.

Proměnné Pro uložení proměnných se také využívá datové struktury `dictionary`, která uchovává dvojice tvořené názvem proměnné a její hodnotou. Pro přidání a modifikaci proměnných slouží v modulu konfigurace funkce `addVar(name)` a `setVar(name, value)`, pro odstranění funkce `removeVar(name)` a pro rekurzivní vyhledání všech proměnných v aplikaci funkce `getAllVars()`.

Směrovací tabulka Směrovací tabulka je opět reprezentována pomocí datové struktury `dictionary`. Záznam v této struktuře je tvořen adresou zdrojového portu a seznamem všech adres cílových portů. Zdrojová adresa je tvaru `instance/port`, cílová adresa má tvar `uzel/instance/port`, respektive `instance/port`, pokud se jedná o směrovací tabulku kompozitu. Pro přidání a odstranění záznamu ze směrovací tabulky jsou určeny funkce `addRoute` a `removeRoute`, jejichž parametry je pětice tvořená zdrojovou instancí a portem a cílovým uzlem, instancí a portem. Pro získání tabulky na jedné úrovni hierarchie slouží funkce `getRoutes()` a pro získání záznamů směrovacích tabulek na všech úrovních funkce `getAllRoutes()`.

5.2.2 Modul klient

Modul klient je implementovaný v souboru `clientMQTT.py`. Obstarává komunikaci aplikace s ostatními uzly v systému. Mezi jeho hlavní úkoly patří vytvoření instance MQTT klienta, který pak navazuje spojení s brokerem a řeší odběr a publikování zpráv s daty aplikace.

Tento modul využívá knihovny `paho.mqtt`, která implementuje třídu Client, realizující klientskou část MQTT komunikace. Tato třída umožňuje připojení k brokeru s daným identifikátorem klienta, pro který se používá název uzlu. Třída také umožňuje nastavit callback funkce pro různé události komunikace a pro různé předměty příchozích zpráv, což využívám pro nastavení vlastních reakcí na dané události a na zprávy s daným předmětem.

on_connect callback funkce, která se volá při připojení klienta k brokeru. Funkce na připojení reaguje zaregistrováním odběru zpráv s předmětem `client_id` a s předmětem `client_id/+/+`. `client_id` je název daného uzlu, který je definován uživatelem při samotném spuštění aplikace. Zprávy posílané s předmětem názvu uzlu jsou pak zpracovávány jako zprávy servisní. Na zprávy s předmětem `client_id/+/+`, kde znak `+` slouží jako náhrada za jednu úroveň předmětu [1], se pak pohlíží jako na zprávy aplikační, čili předmět tvoří název cílového funkčního bloku ve formě `client_id/FB_name/FB_port`, kde `FB_name` je název instance cílového FB a `FB_port` název jeho portu. V této funkci se také registruje k předmětu `system`, prostřednictvím kterého o sobě uzly informují ostatní uzly v síti.

5.2.3 Modul HMI

Modul HMI tvoří servisní rozhraní celého distribuovaného řídicího systému. Rozhraní je implementováno jako webový server, který běží na uzlech sítě a je přístupný přes IP adresu daného uzlu. Vzhledem k tomu, že se u zařízeních Raspberry Pi Zero, které slouží jako uzly systému, neočekává, že by při používání byly napojeny na externí monitor, je vzdálený přístup do HMI vhodnou variantou. Webové prostředí je pro toto rozhraní dostatečně přívětivé a není nijak omezené ve své funkcionalitě.

Prostřednictvím HMI je možné sledovat různé aspekty systému, jako jsou informace o jeho uzlech, o funkčních blocích proměnných na daných uzlech atd. Kromě toho ale umožňuje se systémem interagovat přes MQTT zprávy. Jelikož modul HMI běží v rámci runtime platformy, má přístup k modulu klient, takže je schopný publikovat zprávy na ostatní uzly systému.

Implementace webového serveru je realizována za pomocí knihovny `http.server`[1], konkrétně převzetím třídy `BaseHTTPRequestHandler`, která umožňuje definovat vlastní funkce pro zpracování webových požadavků. Modul definuje pomocné funkce pro generování kódu ve formátu HTML[2], který zobrazuje uživateli data v prohlížeči a také obsahuje funkce pro generování MQTT servisních zpráv na základě uživatelových požadavků.

Šablona pro webové stránky HMI je nadefinovaná v souboru `index.html`. Pro stylování stránky jsem využil framework `W3.CSS`[3].

[1] https://docs.python.org/3/library/http.server.html
[2] HTML – HyperText Markup Language
[3] Více informací o frameworku W3.CSS na https://www.w3schools.com/w3css/default.asp.

5.2.4 Hlavní modul

Hlavní modul aplikace se nachází v souboru `runtime.py`. Tento modul se zabývá samotným řízením aplikace, respektive funkčních bloků aplikace. Modul využívá modulů konfigurace a klienta pro vytvoření zázemí pro běh aplikace a pro navázání spojení s brokerem a ostatními uzly systému. Nachází se v něm zpracovávání příchozích servisních i aplikačních zpráv, definice základní třídy pro atomický a kompozitní funkční blok a definice servisních funkčních bloků.

Standardní běh modulu

Při startu aplikace dochází ke konfiguraci aplikace. Zde se využívá modulu konfigurace. Vytváří se instance třídy `Config()`, do které se vloží základní informace o aplikaci (název uzlu, IP adresa MQTT brokeru,...), a dále se vytvoří instance servisních funkčních bloků, které se také uloží do konfigurace aplikace. Poté se modul pomocí modulu klient pokusí navázat spojení s MQTT brokerem. Na tomto spojení závisí činnost celé aplikace a bez něj její běh nemá z pohledu distribuovaného systému žádný význam, proto v případě neúspěšného navázání spojení dochází k ukončení aplikace. V případě, že aplikace úspěšně naváže kontakt s brokerem, modul vytvoří instanci třídy HMI a spustí webový server, který slouží jako servisní uživatelské rozhraní. Nakonec zahájí modul hlavní programovou smyčku, ve které čeká na příchozí zprávy.

Třídy pro funkční bloky

Hlavní modul definuje dvě třídy, které souvisejí s funkčními bloky distribuovaného řídicího systému. Tyto dvě třídy mají za úkol reprezentovat funkční bloky v rámci celé runtime platformy. Poskytují rozhraní, díky kterému je možné nastavovat vstupní porty bloků, provádět jejich vnitřní algoritmus a také publikovat data na své výstupní porty.

Task() Základní třída definující strukturu funkčního bloku. Dělí se na 3 důležité části: inicializační, exekuční a publikační. V inicializační části, kterou realizuje funkce `__init__(self)`, dochází k nastavení základních informací o funkčním bloku, jako je jméno instance této třídy a případně odkaz na kompozitní blok, je-li daný blok jeho součástí. Dále se při inicializaci definují vstupní porty (proměnné) a lokální proměnné. Všechny vstupní proměnné je třeba nadefinovat ve funkci `__init__(self)`, bez toho by nebylo možné do těchto vstupních portů bloku pak směrovat data z ostatních funkčních bloků. Exekuční část je zahájena přiřazením hodnoty do vstupního portu. K tomu slouží funkce `set_port(self,port,payload)`, která na základě hodnoty v parametru `port` přiřadí do příslušné vstupní proměnné třídy hodnotu `payload`. Samotný algoritmus funkčního bloku se poté nachází ve funkci `run(self)`, která pro svůj běh může využívat libovolné další třídní metody. Pro publikování dat na výstupní porty slouží funkce `publish(self,port, payload)`, jejíž parametry jsou název výstupního portu bloku a výstupní hodnota. Zde se však už rozlišuje chování podle typu funkčního bloku. Pokud se jedná o blok, který není součástí žádného kompozitního bloku, pak funkce na základě záznamů ve směrovací tabulce publikuje zprávy s předmětem cílových bloků a jejich vstupních portů. V případě bloku, který je součástí kompozitního bloku, pak dochází ke směrování dat v rámci daného kompozitního bloku podle jeho vnitřní směrovací tabulky.

Při odstraňování instance bloku se volá metoda `delete(self)`, ve které je vhodné instanci řádně ukončit, aby mohla být úspěšně odstraněna.

CompositeBlock(Task) Tato třída, poděděná od třídy `Task`, slouží jako základní třída pro kompozitní funkční bloky. Jednou z podstatných vlastností kompozitních bloků je, že se navenek chovají stejně jako atomické bloky. Tato vlastnost je zaručena právě výše uvedenou dědičností. Vnitřní podstata kompozitního bloku má však svou funkcionalitou mnohem blíže k runtime platformě než k atomickým blokům. Na rozdíl od nich si musí kompozitní blok uchovávat informace o svých vnitřních blocích a jejich propojení. K tomuto účelu využívá stejně jako samotná runtime platforma modul konfigurace, jehož rozhraní už poskytuje to, co kompozitní blok potřebuje. Během inicializace kompozitního bloku dochází k vytvoření vlastní instance třídy `Config` a jinak stejně jako u každého bloku se nastavuje jméno instance a případný odkaz na kompozitní blok, jehož je daný blok součástí. Inicializace dále obsahuje instanciaci vnitřních funkčních bloků daného uzlu, jejichž seznam je ve forme dvojice `(ins,name)` předáván jako parametr při jeho aktivaci. Stejně tak dochází k nastavení vnitřních proměnných a propojení vnitřních bloků formou směrovací tabulky. Jak vnitřní proměnné, tak propojení bloků se opět předává při aktivaci daného kompozitního bloku. Při volání kompozitního bloku (na jeho vnitřní port jsou předána data) si blok na základě vstupního portu vyhledá ve směrovací tabulce port vnitřního bloku, který je na něj namapován, tomu předá vstupní data a spustí jeho činnost volání funkce `run()`. Pro účely publikování dat mezi vnitřními bloky definuje třída funkci `route(self,port,payload)`, která vyhledává cílové bloky, předá jim hodnoty na vstupní porty a spustí jejich činnost. Při odstranění instance kompozitního bloku jsou odstraněny všechny jeho vnitřní bloky, což implmentuje metodata `delete(self)`.

Servisní funkční bloky

Hlavní modul definuje několik servisních funkčních bloků, které slouží ke správě aplikace. Tyto bloky úzce souvisí se servisními MQTT zprávami, tak jak jsou navrhnuty v kapitole [4]. Na servisní bloky se pohlíží jako na atomické funkční bloky, tudíž jsou definované jako třídy, které dědí od třídy `Task`. Každý blok má vstupní port `payload`, na kterém očekává data odpovídající obsahu servisní MQTT zprávy. Funkcionalita bloků je pak implementována ve funkci `run()`.

Load(Task) Načítá definici (zdrojový kód) uživatelského funkčního bloku. Na vstupním portu očekává definici třídy, která uživatelský FB realizuje. Načtení pak probíhá pomocí vestavěné funkce `exec()`, která dynamicky načte definici do běžící aplikace. Aby nedocházelo k opětovnému načítaní stejné třídy v případě opakovaného příchodu servisní zprávy `Load`, ukládá informaci o načtené třídě do konfigurace aplikace.

Unload(Task) Odstraňuje třídu funkčního bloku z runtime platformy. Vstupní port obsahuje název třídy, která se má z platformy odstranit. Definice třídy je pak odstraněna pomocí funkce `del`, a stejně tak je odstraněn i záznam ze seznamu tříd.

Activate(Task) Vytváří instanci funkčního bloku. Na vstupním portu očekává název třídy, název nové instance té třídy a parametry pro instanciaci. Blok kontroluje, aby název instance nebyl v konfliktu s ostatními instancemi FB, a také zda pro danou třídu existuje její definice. Po úspěšných kontrolách vytvoří novou instanci a uloží ji do konfigurace aplikace. V případě aktivace kompozitního bloku je jako název třídy očekávána hodnota `CompositeBlock`, čili třída, která definuje kompozitní bloky (viz výše v sekci [5.2.4]).

Remove(Task) Odebírá instanci funkčního bloku z aplikace. Na vstupním portu očekává název instance, kterou má odstranit. Při odstranění je instance odebrána z konfigurace aplikace.

AddRoute(Task) Přidá záznam do směrovací tabulky. Na vstupním portu očekává pětici identifikující zdrojovou instanci a její port, respektive zdrojovou proměnnou, a cílový uzel, instanci a její port, respektive cílový uzel a proměnnou. Nový záznam ukládá do konfigurace aplikace.

RemoveRoute(Task) Odebírá záznam ze směrovací tabulky. Data na vstupním portu viz servisní blok `AddRoute(Task)`

AddVar(Task) Přidá novou proměnnou do aplikace. Na vstupní portu očekává název nové proměnné. Kontroluje, zdali už proměnná s daným názvem v aplikaci neexistuje. Novou proměnnou ukládá do konfigurace aplikace s hodnotou `None`.

SetVar(Task) Nastavuje hodnotu proměnné aplikace. Na vstupním portu očekává dvojici obsahující název proměnné a hodnotu, na kterou se má nastavit. Aktualizuje hodnotu proměnné v konfiguraci aplikace.

RemoveVar(Task) Odstraňuje proměnnou z aplikace. Vstupní port musí obsahovat název proměnné, která se má odstranit. Blok vymaže záznam o proměnné, který je uložený v konfiguraci aplikace.

Status(Task) Získává a posílá data o daném uzlu ostatním uzlům. V případě požadavku typu `Get` požádá konfigurační modul o informace, které má uložené, a odešle je v servisní zprávě `Status` typu `Post` na uzel, který o informace požádal. Pokud přijme zprávu s daty, uloží je do speciální proměnné modulu konfigurace.

Uživatelské funkční bloky

Definice uživatelských funkčních bloků má pevně danou strukturu podle třídy `Task`. Pro vytvoření vlastního (uživatelského) bloku je třeba vytvořit nový soubor ve složce `SpecialFB` ve formátu `FB_name.py`, kde `FB_name` odpovídá názvu vytvářeného bloku. V tomto souboru je FB definován pomocí třídy s názvem `FB_name`, která musí dědit od třídy `Task`. Třída musí implementovat funkci `__init__`, kde definuje případné vstupní porty nebo vnitřní

proměnné, a funkci `run`, kde implementuje vnitřní algoritmus funkčního bloku. Na začátku souboru je možné importovat potřebné knihovny pro daný funkční blok. Pro publikování dat na výstupní porty se používá výhradně funkce `publish` ze třídy `Task`. Při zpracovávání dat na vstupních portech je třeba počítat s faktem, že předávaná data jsou typu `bytes`.

Výpis 5.1 znázorňuje příklad uživatelského funkčního bloku v jazyce Python.

```
class ANDFB(Task):
  def __init__(self):
    self.in1 = "False"
    self.in2 = "False"

  def str2bool(self, _str):
    return _str == "True"

  def run(self):
    if isinstance(self.in1, str):
      self.in1 = self.str2bool(self.in1)
    if isinstance(self.in2, str):
      self.in2 = self.str2bool(self.in2)

    self.publish("out", self.in1 and self.in2)
```

Výpis 5.1: Příklad uživatelského funkčního bloku implementovaného v jazyce Python.

Zpracování příchozích zpráv

Nezanedbatelnou částí hlavního modulu je zpracovávání přijatých zpráv. Zprávy, které přijímá modul klient, jsou na základě jejich předmětu děleny na zprávy servisní (zprávy s předmětem shodujícím se s názvem uzlu), uživatelské (zprávy s předmětem `node_name/+/+`, viz 5.2.2) a systémové (s předmětem `system`). Modul klient umožňuje nastavit pro každý typ zprávy jinou callback funkci, které jsou popsané níže.

node_exec() Callback funkce pro zpracování servisních zpráv. Na základě dekódovaného obsahu zprávy, který je ve formátu `{Action,Payload}`, viz sekce 4.5, získá odpovídající instanci servisního bloku, předá ji na vstupní port hodnotu `Payload` a spustí ji funkcí `run`.

task_exec() Callback funkce pro zpracování uživatelských zpráv. Zde je název cílové instance funkčního bloku zakódovaný v předmětu zprávy a předávaná hodnota se nachází přímo v těle zprávy. K instanci se dá přistoupit pomocí modulu konfigurace, který zároveň kontroluje existenci instance v aplikaci. V případě, že na uzlu neexistuje adresovaná instance, je tato zpráva ignorována.

system_exec() Callback funkce pro zpracování systémových zpráv. Systémové zprávy slouží pro identifikaci jednotlivých uzlů systému mezi sebou. Obsahem zprávy je název uzlu, který zprávu publikoval. Zprávu přijímá každý uzel v síti a informaci o názvu zdrojového uzlu této zprávy si ukládá do své konfigurace. V případě, že už informaci o daném uzlu má, pak tuto zprávu ignoruje.

Zaznamenávání činnosti systému

Při běhu runtime platformy dochází k zaznamenávání (neboli k tzv. logování) důležitých operací. Tyto záznamy se ukládají do souboru `runtime.log` a zároveň jsou publikovány pod předmětem `logging`. Tímto způsobem je možné sledovat, jestli při startu či běhu platformy nedošlo k nějakým závažným chybám, které by znemožnily korektní běh systému. Zaznamenávají se i operace servisních funkčních bloků, konkrétně zdali operace proběhla úspěšně či neúspěšně.

5.3 Inicializační skript

Jeden z výstupů této práce je skript, který funguje jako nástroj pro usnadnění nahrávání uživatelských aplikací na uzly distribuovaného řídicího systému. Jedná se o nástroj, který zpracovává program nadefinovaný v jazyce ST (více se o programu v jazyce ST dočtete v kapitole [6]) a generuje MQTT servisní zprávy, které instalují program na jednotlivé uzly DCS. Skript prakticky inicializuje aplikaci v rámci systému neboli provede deployment (česky rozestavení) aplikace na uzly. Skript postupně analyzuje celý vstupní soubor a ukládá si průběžně získané informace ohledně definice funkčních bloků, proměnných a programu a také o jejich vzájemném propojení. Na základě povinných proměnných je schopný rozlišit, které FB se mají nainstalovat na které uzly systému, jaké instance na nich mají vzniknout atd. Po úspěšné analýze se skript připojí na MQTT broker, jehož IP adresa je předána v parametru skriptu, a začne publikovat servisní zprávy na uzly sítě. Pro analýzu skript využívá standardních funkcí jazyka Python a také regulárních výrazu. V průběhu zpracovávání vstupního souboru si na příslušných místech ukládá servisní zprávy, které je třeba poslat do systému, aby se aplikace řádně nainstalovala. Zprávy začne odesílat, až když zpracuje celý soubor. Při tvorbě servisní zprávy typu `Load`, která slouží k nainstalování uživatelských funkčních bloků na uzly, načítá zdrojové kódy ze složky SpecialFB.

Kapitola 6

Tvorba aplikací pro realizovanou platformu a její testování

6.1 Tvorba aplikace

Uživatelská aplikace je definovaná ve své podstatě pouze sekvencí několika servisních zpráv zaslaných na MQTT broker systému. Tohoto je možné dosáhnout buď použitím inicializačního skriptu, nebo obecně vytvořením a zasláním zpráv, prostřednictvím libovolného MQTT klienta, na broker, který zprávy publikuje na cílové uzly. Vyvíjet a navrhovat systém jenom pomocí sekvence servisních zpráv je však značně komplikované a nepřehledné a jakékoliv případné modifikace by se dodatečně implementovaly velice složitě. Jako potenciální řešení tohoto problému může slouží servisní HMI, které je v runtime platformě vestavěné, ale ani to neumožňuje definovat aplikaci přehledně, pouze by usnadnilo publikaci zpráv do systému. Jako řešení a nabídnutí možnosti efektivní definice DSC jsem se proto rozhodl využít již existujícího nástroje OpenPLC Editor[1], který umožňuje psát PLC programy mimo jiné v jazycích Structured Text a Function Block Diagram a generovat výstup čistě ve formátu Structured. Vzhled k důrazu, který norma IEC 61131-3 klade na strukturu PLC programů, je tento formát vhodný pro definici aplikace pro mnou vyvíjený systém, a proto jsem se rozhodl jej použít jako hlavní z možných způsobů definic uživatelských aplikací.

Definice uživatelské aplikace v nástroji OpenPLC Editor

Velkou výhodou tohoto editoru je možnost grafického programovaní. To umožňuje přehledně navrhnout celý systém za použití grafických komponent, které reprezentují funkční bloky, funkce, atd. Je však třeba mít na paměti, že editor slouží k vytváření PLC programu a nikdy nebyl určen k definici aplikací pro mnou vyvíjený systém. Rozhodl jsem se však využít možností tohoto softwaru, jelikož při jeho specifickém použití lze z něj vygenerovat definici aplikace v jazyce ST, jenž dokáže zpracovat inicializační skript.

DCS, který je vytvářen v tomto projektu, podporuje pouze dva typy POU[2], a sice: programy a funkční bloky. Program v kontextu tohoto projektu reprezentuje uživatelskou

[1] https://www.openplcproject.com/plcopen-editor/

[2] POU – Program Organization Unit

aplikaci. Funkční bloky pak reprezentují jak funkční bloky, tak i funkce. Při definici aplikace v OpenPLC editoru je tedy nutné používat pouze tyto dvě komponenty.

Program Úkolem této komponenty je definovat celou uživatelskou aplikace. Ta se skládá z pohledu funkcionality z propojení instancí funkčních bloků a proměnných a z pohledu DCS ze vzájemně propojených koncových uzlů. Zatímco funkcionalitu systému lze definovat snadno pomocí prostředků, které editor nabízí, logiku rozložení funkčních bloků na uzly je třeba zahrnout do definice programu. K tomu slouží speciální rezervované proměnné. Program musí obsahovat lokální proměnnou `_nodes`, ve které je uložený seznam uzlů systému, respektive jejich názvů, ve tvaru `["node1","node2",...]`. Pro každý uzel pak musí být vytvořena lokální proměnná s názvem daného uzlu, jejíž hodnota obsahuje seznam všech instancí a proměnných, které se na daném uzlu mají nacházet. Program dále může obsahovat libovolný počet dalších lokálních proměnných a samotné instance funkčních bloků, které realizují funkcionalitu systému. Pro tyto účely je vhodné definovat program v jazyce FBD. Ve výpisu 6.1 se nachází definice programu `DemoApp`.

```
1  PROGRAM DemoApp
2    VAR
3      _nodes : STRING := '["rpi1", "rpi2"]';
4      rpi1 : STRING := '["LuxSenzorCFB0", "LuxLimit", "LTFB0", "LEDFB0"]';
5      rpi2 : STRING := '["SwitchFB0", "ActivatorFB0", "TrigBothFB0", "PIRSenzorCFB0", "ANDFB0
       "]';
6      LuxLimit : STRING := '300';
7      ActivatorFB0 : ActivatorFB;
8      SwitchFB0 : SwitchFB;
9      LTFB0 : LTFB;
10     LEDFB0 : LEDFB;
11     LuxSenzorCFB0 : LuxSenzorCFB;
12     TrigBothFB0 : TrigBothFB;
13     PIRSenzorCFB0 : PIRSenzorCFB;
14     ANDFB0 : ANDFB;
15   END_VAR
16
17   LuxSenzorCFB0();
18   LTFB0(in1 := LuxSenzorCFB0.out, in2 := LuxLimit);
19   TrigBothFB0(in1 := LTFB0.out);
20   PIRSenzorCFB0();
21   ANDFB0(in1 := TrigBothFB0.out, in2 := PIRSenzorCFB0.out);
22   SwitchFB0();
23   ActivatorFB0(in_senzor := ANDFB0.out, in_switch := SwitchFB0.out);
24   LEDFB0(in1 := ActivatorFB0.out);
25 END_PROGRAM
```

Výpis 6.1: Ukázka definice programu v jazyce ST.

Funkční bloky Stejně jako u definice programu i u definice funkčních bloků pro DCS v této práci je třeba dodržet specifickou strukturu. Každý funkční blok je třeba vytvořit jako vlastní uživatelsky nadefinovaný blok s povinnou lokální proměnnou `class`, jejíž hodnota udává jméno třídy, která definuje blok v jazyce Python. Vzhledem k omezení editoru, který neumožňuje definovat funkční blok s prázdným tělem, je třeba definovat funkční blok v jazyce ST a v jeho děle inicializovat proměnnou `class` s příslušnou hodnotou. Vnitřní algoritmus funkčního bloku se v editoru nijak nedefinuje, jelikož ten je implementován v odpovídající třídě, je však třeba nadefinovat vstupní a výstupní proměnné, aby bylo možné používat

blok při definici programu. Výjimkou u definice bloků jsou bloky kompozitní, které je možné definovat graficky v jazyce FBD podobně jako program. Kompozitní blok však musí mít povinně lokální proměnnou `BlockType` s hodnotou `composite`, čím se odlišuje od atomických FB. Ve výpisu 6.2 se nachází příklad definice funkčního bloku v jazyce ST.

```
FUNCTION_BLOCK ORFB
  VAR_INPUT
    in1 : BOOL;
    in2 : BOOL;
  END_VAR
  VAR_OUTPUT
    out : BOOL;
  END_VAR
  VAR
    class : STRING;
  END_VAR

  class := 'ORFB';
END_FUNCTION_BLOCK
```

Výpis 6.2: Ukázka definice funkčního bloku v jazyce ST.

Existence speciálních proměnných pro definici programu a funkčních bloků mě vedla k vytvoření seznamu rezervovaných názvů proměnných: `_nodes`, `BlockType`, `class`.

Funkcí editoru `vygenerovat program pro OpenPLC runtime` vznikne soubor v jazyce ST, který pak slouží jako vstup do inicializačního skriptu.

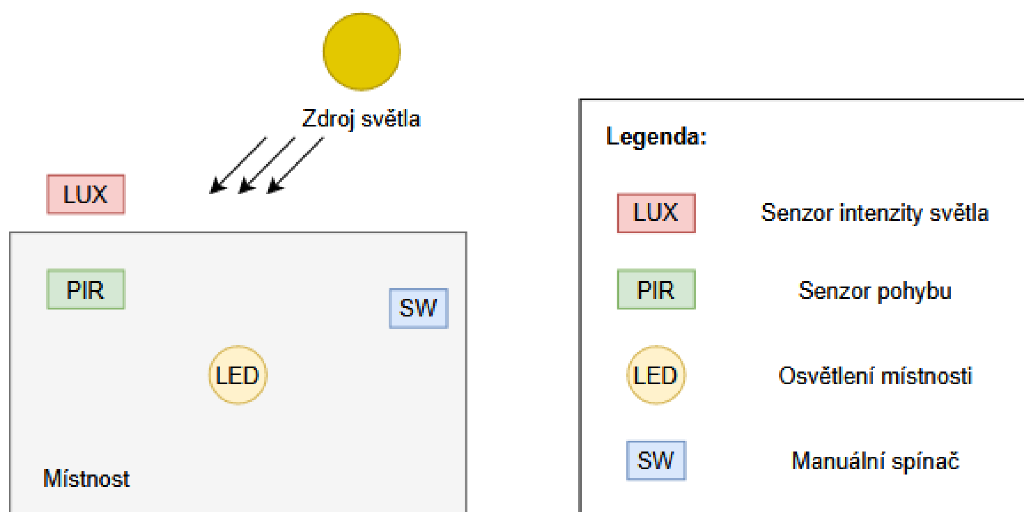
6.2 Demo aplikace

Vytvořený systém byl testovaný na demo aplikaci, která se snaží zároveň demonstrovat i jeho možnosti použití. Demo aplikace řeší problém osvětlení místností na základě vnějších faktorů, jako je intenzita okolního světla, pohyb v místnosti a manuální sepnutí spínače.

6.2.1 Popis problému

Jedním z cílů demo aplikace je demonstrovat využití vyvinutého systému na reálné situaci. Pro tento účel jsem se rozhodl řešit problém automatického osvětlení místnosti, které je ovlivňováno více různými faktory, které nyní popíšu podrobněji.

Intenzita okolního světla Vliv tohoto faktoru hraje při řešení problému osvětlení podstatnou roli. Jednou z myšlenek pro automatické osvětlení je určit situace, kdy je potřeba světlo zapínat. Přirozenou odpovědí na tuto otázku je, když je tma. Intenzita světla je měřena v jednotkách LUX a existují přístroje pro její měření. Na základě přístupu světla do osvětlované místnosti lze tedy určit práh intenzity světla, který určuje, kdy je v ní dostatečné množství světla a tudíž není potřeba místnost uměle osvětlovat, a naopak kdy je v ní světla málo a je třeba místnost dodatečně osvětlit. Stanovení prahu pro místnost vychází z předpokladu, že je intenzita světla snímána mimo místnost samotnou, ideálně na místě, kde intenzitu světla ovlivňují pouze stálé zdroje světla (jako je například Slunce.)



Obrázek 6.1: Návrh demo aplikace.

Detekovaný pohyb v místnosti Dalším významným faktorem pro automatické osvětlení místnosti je, zdali je v ní někdo přítomen. Sepnutí světla v prázdné místnosti by bylo neekonomické a zbytečné, ale je žádoucí, když se v místnosti někdo pohybuje. Přítomnost osob v místnosti umožňují detekovat detektory pohybu.

Manualní sepnutí spínače V situacích, které není možné ošetřit pomocí dvou předchozích faktorů, je třeba umožnit ovládání světla v místnosti i manuálně, prostřednictvím klasického spínače. Spínač v tomto případě způsobuje změnu stavu osvětlení v místnosti nezávisle na okolních faktorech.

Struktura systému

Automatické osvětlení místnosti je řízeno těmito třemi faktory. Demo aplikace se tedy skládá ze dvou senzorů, pohybu a intenzity světla, z jednoho spínače a ze zdroje osvětlení v místnosti. Tuto skutečnost demonstruje obrázek 6.1.

Pro demonstrační účely této aplikace jsem využil dvou zařízení Raspberry Pi Zero, jako hlavních uzlů distribuovaného řídicího systému.

Požadavky na řešení problému

Na základě výše uvedených faktorů jsem definoval následující požadavky na systém poskytující automatické osvětlení, čili situace, ve kterých se má osvětlení aktivovat, respektive deaktivovat.

- Osvětlení se aktivuje, když intenzita světla klesne pod předem stanovený práh a v místnosti je detekován pohyb.

- Osvětlení se deaktivuje, pokud intenzita světla stoupne nad předem stanovený práh, nebo se v místnosti žádný pohyb nedetekuje.
- Pokud je osvětlení aktivní a spínač osvětlení je sepnut, pak se osvětlení deaktivuje a je v tomto stavu do té doby, než je spínač znovu sepnut.
- Pokud osvětlení není aktivní a spínač je sepnut, pak se osvětlení aktivuje a je v tomto stavu, dokud není spínač znovu sepnut.

Cílem demo aplikace je vytvořit distribuovaný systém, který tyto požadavky splňuje.

6.2.2 Hardware využitý v demo aplikaci

Demo aplikace běží jako distribuovaný řídicí systém, jehož uzly tvoří zařízení Raspberry Pi Zero, na které jsou napojena všechna potřebná periferní zařízení, jež slouží jako vstupy a výstupy aplikace. Ze sekce 6.2.1 vyplývá, jaká periferní zařízení jsou potřeba pro správný běh aplikace. Zařízení použitá pro tuto aplikaci jsou popsána v následujících sekcích.

Senzor intenzity světla Pro monitorování okolního osvětlení, na základě kterého se určuje, zdali je třeba aktivovat osvětlení v místnosti, jsem použil modul intenzity světla GY-302 BH1750. Jedná se o digitální modul, který je určený pro zapojení s mikrokontroléry. Jeho rozsah je od 0–65535 lx, což je pro účely demo aplikace dostačující, jelikož typická denní intenzita osvětlení se obvykle pohybuje mezi 100–10000 lx.[3] Modul komunikuje skrze rozhraní I2C, takže je možné jej napojit na Raspberry Pi Zero. Výstupem z tohoto senzoru je tedy intenzita snímaného světla v jednotkách LUX. Senzor je na obrázku 6.2.

Detektor pohybu Pro detekci pohybu v místnosti jsem použil senzor PIR HC SR501. Jedná se o pasivní pyroelektrický infračervený senzor s nastavitelnou citlivostí a časem sepnutí. Senzor umožňuje detekovat pohyb od vzdálenosti 3 metrů až po 7 metrů. Senzor komunikuje s Raspberry Pi Zero skrze univerzální vstupně výstupní pin tak, že při detekci pohybu jej nastaví na vysokou úroveň. Senzor je na obrázku 6.3.

Spínač Jako spínač slouží v demo aplikaci jednoduchý mikrospínač, který simuluje činnost klasických spínačů světel v místnostech.

Světelný zdroj Pro účely demo aplikace jsem jako světelný zdroj, který znázorňuje osvětlení místnosti použil klasickou LED diodu.

Ostatní součástky Pro úplný výpis použitého hardwaru v demo aplikaci je třeba ještě uvést několik dalších součástek, jako jsou spojovací kabely, nepájivé pole (breadboard), několik rezistorů a kapacitor pro spolehlivější detekci stisku spínače.

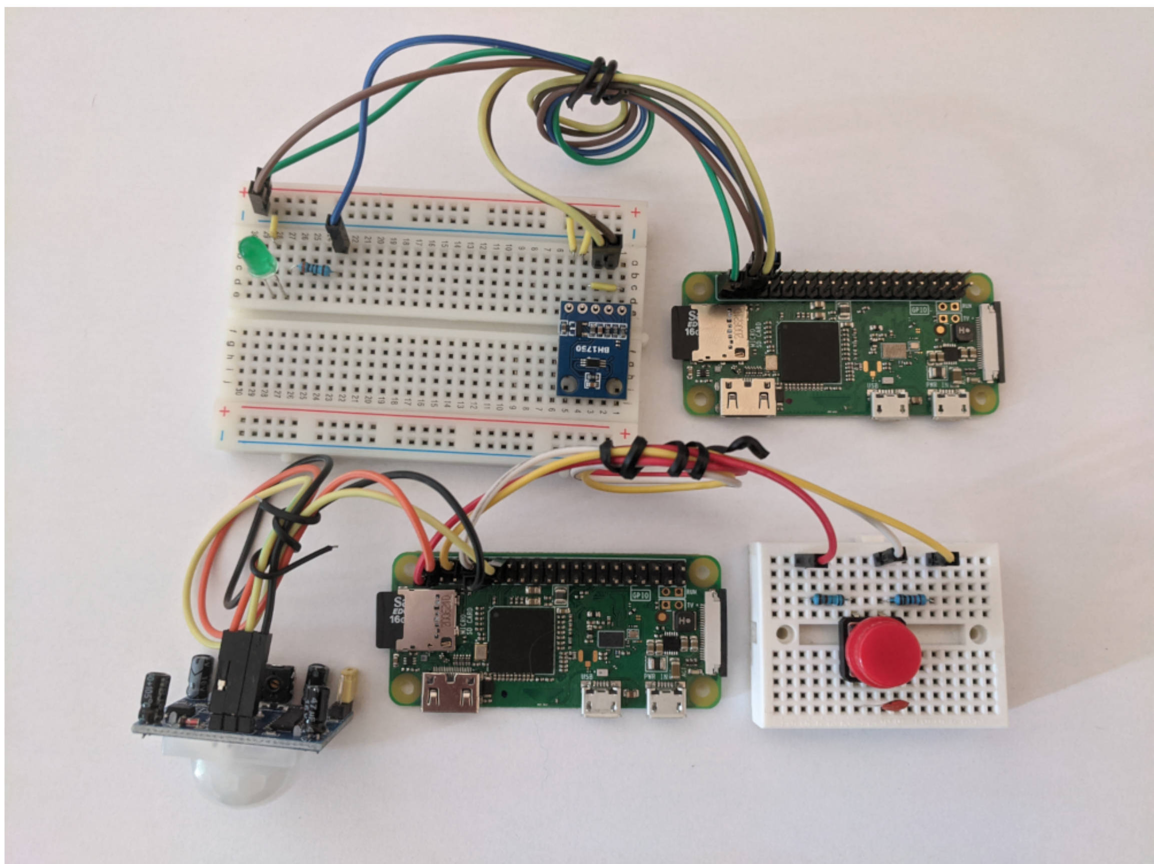
Zapojení všech hardwarových komponent je znázorněno na obrázku 6.4. Toto zapojení pouze simuluje reálnou situaci, ve které by senzory byly umístěny na vhodných místech, místo LED

[3] https://cs.wikipedia.org/wiki/Lux_(jednotka)

Obrázek 6.2: Senzor PIR HC SR501.

Obrázek 6.3: GY-302 BH1750

diody by bylo Raspberry Pi napojené na reálný zdroj světla v místnosti, stejně tak i spínač v tomto zapojení pouze nahrazuje funkci klasických spínačů světel. Pro účely demonstrace mého distribuovaného systému je však toto zapojení naprosto dostačující, jelikož logika aplikace by byla stejná jako v reálné situaci.



Obrázek 6.4: Fyzické zapojení hardwaru. Vlevo nahoře je v nepájivém poli připojena LED dioda a senzor GY-302 BH1750 s jedním Raspberry Pi Zero a dole je senzor PIR HC SR501 a spínač propojený s druhým Raspberry Pi Zero.

6.2.3 Implementace demo aplikace

Implementace demo aplikace byla provedena na základě návrhu a požadavků ze sekce [6.2.1]. Diagram funkčních bloků byl nadefinován v nástroji OpenPLC editor a jednotlivé funkční bloky v jazyce Python.

Diagram funkčních bloků

Nástroj OpenPLC editor umožňuje snadnou definici uživatelské aplikace pro distribuované řídicí systémy, jelikož nabízí nástroje pro grafické definování programu. Při tvorbě diagramu funkčních bloků pro demo aplikaci jsem postupoval následujícím postupem. V prvé řadě bylo třeba identifikovat funkční bloky, a poté jejich propojení. K tomu jsem využíval techniky dekompozice hlavního problému na menší podproblémy, dokud jsem se nedostal až k dále nedělitelným problémům. Tímto postupem jsem nadefinoval několik základních funkčních bloků:

LuxSenzorCFB Jedná se o kompozitní funkční blok, který zapouzdřuje chování senzoru intenzity světla. Výstupem tohoto bloku je hodnota snímané intenzity v jednotce LUX. Výstup z tohoto bloku je periodický.

PIRSenzorCFB Tento další kompozitní funkční blok definuje chování senzoru pohybu. Blok generuje na svůj výstupní port logickou hodnotu `True`, když je v místnosti detekován pohyb. Po uplynutí určité doby, ve které není detekován žádný pohyb, generuje logickou hodnotu `False`.

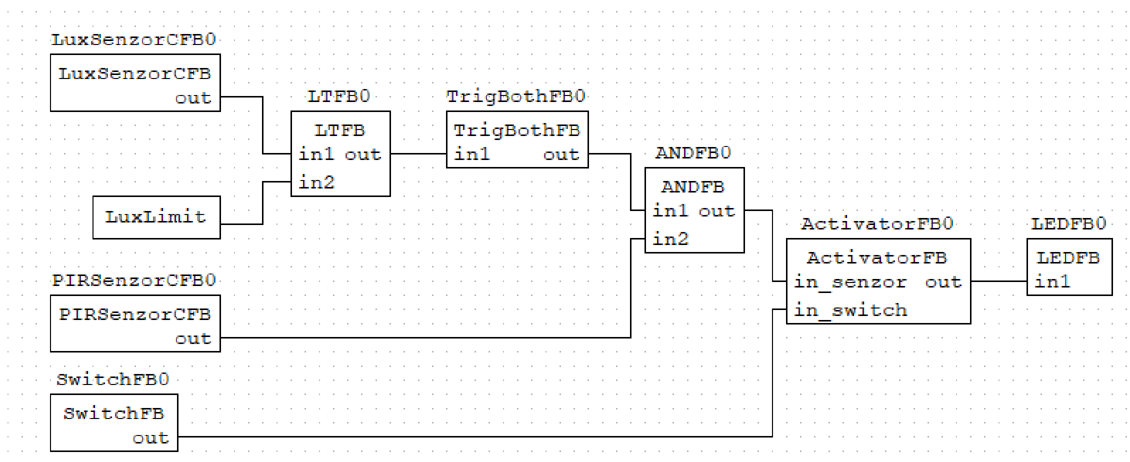
SwitchFB Atomický funkční blok, popisující chování spínače. Na svůj výstup generuje logickou hodnotu `True` vždy, když je spínač sepnut.

ActivatorFB Úkolem tohoto atomického funkčního bloku je zpracovat vstupy ze senzorů a ze spínače a na jejich základě generovat výstup určující stav osvětlení místnosti.

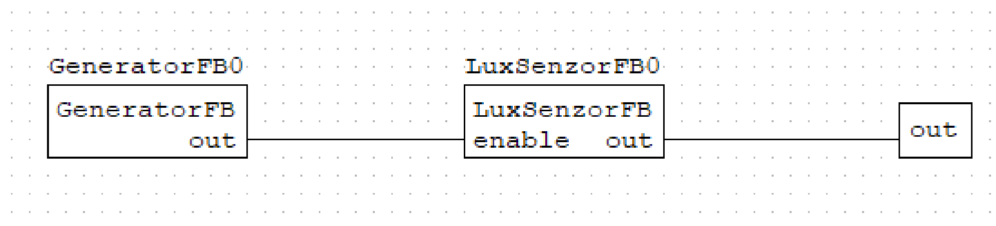
LEDFB Pro ovládání LED diody znázorňující osvětlení místnosti slouží tento atomický funkční blok. Na základě vstupu, kterým je logická hodnota `True` nebo `False`, ji aktivuje, respektive deaktivuje.

Tyto základní funkční bloky jsem propojil podle požadavků v sekci 6.2.1 s pomocí několika dalších pomocných funkčních bloků (viz 6.5), mezi kterými stojí za zmínku blok `TrigBothFB`, který na vstupním portu očekává logické hodnoty, a když dojde ke jejich změně, tak na výstup vygeneruje novou hodnotu. Tím dochází k potlačení výstupů se stejnými hodnotami.

Definovaní kompozitních bloků pak vycházelo z požadavků na jejich chování. Pro blok `LuxSenzorCFB` bylo třeba nadefinovat blok, který čte hodnoty intenzity světla ze senzoru, a blok, který toto čtení periodicky volá. K tomu jsem vytvořil bloky `LuxSenzorFB` a `GeneratorFB` a propojil je, jak je znázorněno na obrázku 6.6.

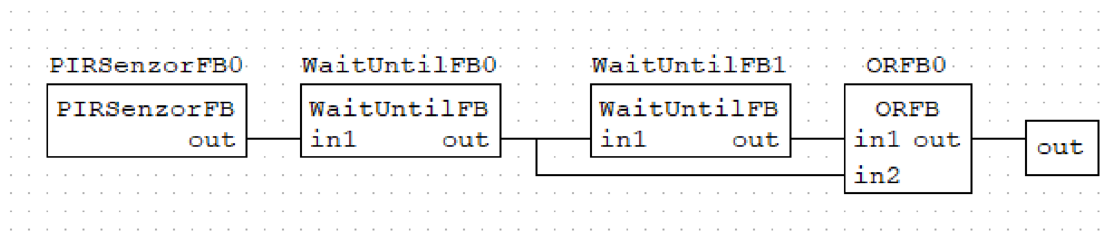


Obrázek 6.5: Diagram funkčních bloků demo aplikace.



Obrázek 6.6: Propojení vnitřních bloků kompozitního bloku LuxSenzorCFB.

Kompozitní blok `PIRSenzorCFB` se skládá z bloků `PIRSenzorFB`, jež je přímo propojený na PIR senzor a který generuje na výstup hodnotu na senzoru. Tento blok je napojený na `WaitUntilFB`, který generuje na výstup hodnotu `True` tehdy, když od doby, kdy vstup přišla hodnota `True`, do doby určené limitem, zaznamená na vstupu další hodnotu `True`. Tento blok slouží jako ověření detekce pohybového senzoru. Dále následuje opět blok `WaitUntilFB`, který je nastavený na dobu, během které se očekává přítomnost osob v místnosti. Při každém dalším detekovaném pohybu se doba opět prodlouží a na výstup se generuje hodnota `True`. Pokud však do té doby žádný další pohyb není detekovaný, předpokládá se, že osoba opustila místnost, a na výstup se generuje hodnota `False`. Propojení bloků je znázorněno v obrázku 6.7.



Obrázek 6.7: Propojení vnitřních bloků kompozitního bloku PIRSenzorCFB.

Pomocí funkce editoru OpenPLC jsem vygeneroval definici demo aplikace ve formátu structured text, která slouží jako vstup do inicializačního skriptu.

Zavedení demo aplikace do systému

Pro nahrání demo aplikace do distribuovaného řídicího systému, který se v tomto případě skládá ze dvou uzlů, jsem použil inicializační skript. Ten po zpracování souboru s definicí aplikace vygeneroval příslušné servisní zprávy pro dané uzly systému. Výsledek této operace je patrný z logových souborů, které vygenerovaly runtime platformy na obou uzlech, viz 6.3, kde je výstup logového souboru jednoho z uzlů.

```
2021-05-02 19:07:47,764 INFO    Runtime application started.
2021-05-02 19:07:47,823 INFO    Connected to MQTT broker with IP address: 192.168.0.136
2021-05-02 19:07:47,842 INFO    HMI:    Server started http://0.0.0.0:8080
2021-05-02 19:08:02,838 INFO    APP:    New node found: rpi2.
2021-05-02 19:08:05,744 INFO    LOAD:   Class LuxSenzorFB successfully loaded.
2021-05-02 19:08:05,756 INFO    LOAD:   Class GeneratorFB successfully loaded.
2021-05-02 19:08:05,764 INFO    LOAD:   Class LTFB successfully loaded.
2021-05-02 19:08:05,774 INFO    LOAD:   Class LEDFB successfully loaded.
2021-05-02 19:08:05,815 INFO    ACTIVATE:       Task LuxSenzorFB0 was successfully
  activated.
2021-05-02 19:08:05,823 INFO    ACTIVATE:       Task GeneratorFB0 was successfully
  activated.
2021-05-02 19:08:05,834 INFO    SETVAR: Value of variable BlockType was successfully set
  to 'composite'.
2021-05-02 19:08:05,836 INFO    ACTIVATE:       Task LuxSenzorCFB0 was successfully
  activated.
2021-05-02 19:08:05,896 INFO    ACTIVATE:       Task LTFB0 was successfully activated.
2021-05-02 19:08:05,908 INFO    ACTIVATE:       Task LEDFB0 was successfully activated.
2021-05-02 19:08:05,916 INFO    ADDVAR: Variable LuxLimit successfully added.
2021-05-02 19:08:05,925 INFO    ADDROUTE:       Route from [LuxSenzorCFB0/out] to [rpi1/
  LTFB0/in1] successfully added.
2021-05-02 19:08:05,935 INFO    ADDROUTE:       Route from [LuxLimit] to [rpi1/LTFB0/in2]
   successfully added.
2021-05-02 19:08:05,944 INFO    ADDROUTE:       Route from [LTFB0/out] to [rpi2/
  TrigBothFB0/in1] successfully added.
2021-05-02 19:08:05,987 INFO    SETVAR: Value of variable LuxLimit was successfully set
  to '300'.
```

Výpis 6.3: Utržek z logovacího souboru znázorňující instalaci aplikace na uzlu rpi1

Správa demo aplikace

Ve chvíli, kdy je aplikace zavedena do systému, je možné monitorovat dění v systému v servisním uživatelském rozhraní. Pro zpřístupnění tohoto rozhraní je třeba znát IP adresu jednoho z uzlů, jelikož na té běží webový server, který rozhraní poskytuje. Na záložce `Data Monitoring` jsou pak vidět hodnoty vstupních a výstupních portů funkčních bloků v reálném čase, viz obrázek 6.8.

Prostřednictvím servisního rozhraní je aplikaci možné modifikovat za jejího běhu.

6.2.4 Node-RED

Alternativou ke vzdálenému monitorování systému skrze servisní HMI je použití existujícího nástroje Node-RED[4] vyvinutého společností IBM. Cílem tohoto nástroje je umožnit vizuální

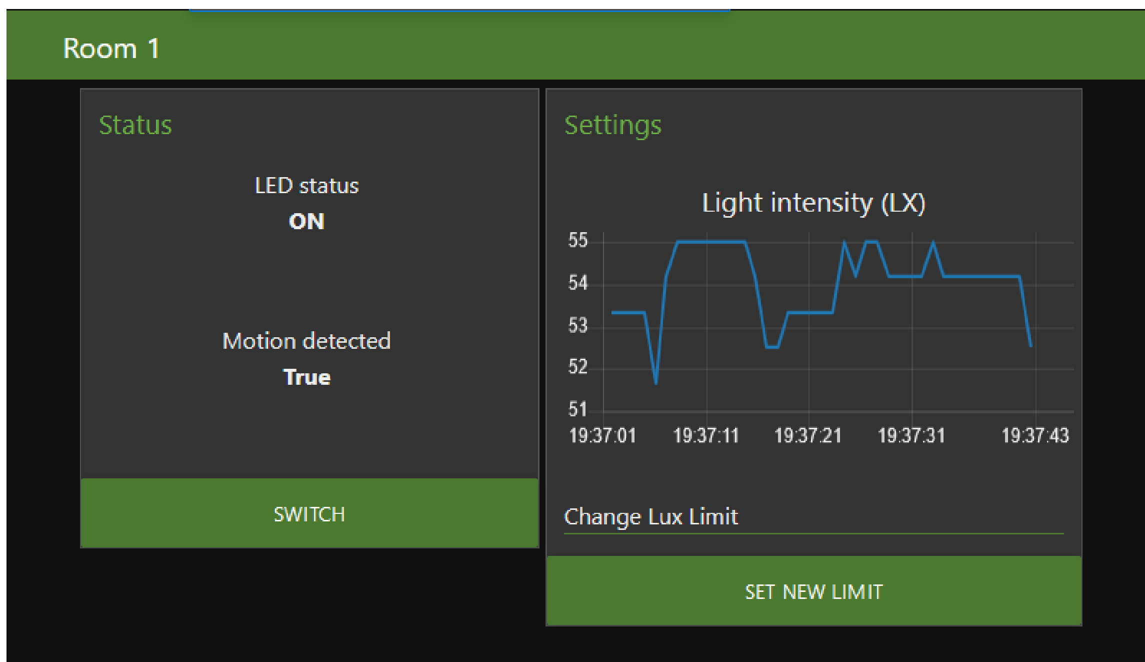
[4] https://nodered.org/

HMI

Nodes | Classes | Instances | Variables | Routes | Send MQTT message | Data Monitoring

Data monitoring				
ANDFB0	out: True	in1: True	in2: True	
ActivatorFB0	is_manual: False	out: True	in_senzor: True	in_switch: True
LEDFB0	in1: True			
LTFB0	in2: 300	in1: 45.83	out: True	
LuxSenzorCFB0	out: 45.83			
LuxSenzorCFB0/GeneratorFB0	out: True			
LuxSenzorCFB0/LuxSenzorFB0	enable: True	out: 45.83		
PIRSenzorCFB0	out: True			
PIRSenzorCFB0/ORFB0	in2: True	in1: True	out: True	
PIRSenzorCFB0/PIRSenzorFB0	out: True			
PIRSenzorCFB0/WaitUntilFB0	in1: True	out: True		
PIRSenzorCFB0/WaitUntilFB1	in1: True	out: True		
SwitchFB0	out: True			
TrigBothFB0	in1: True	out: True		

Obrázek 6.8: Ukázka sběru dat v reálném čase v servisním uživatelském rozhraní demo aplikace.

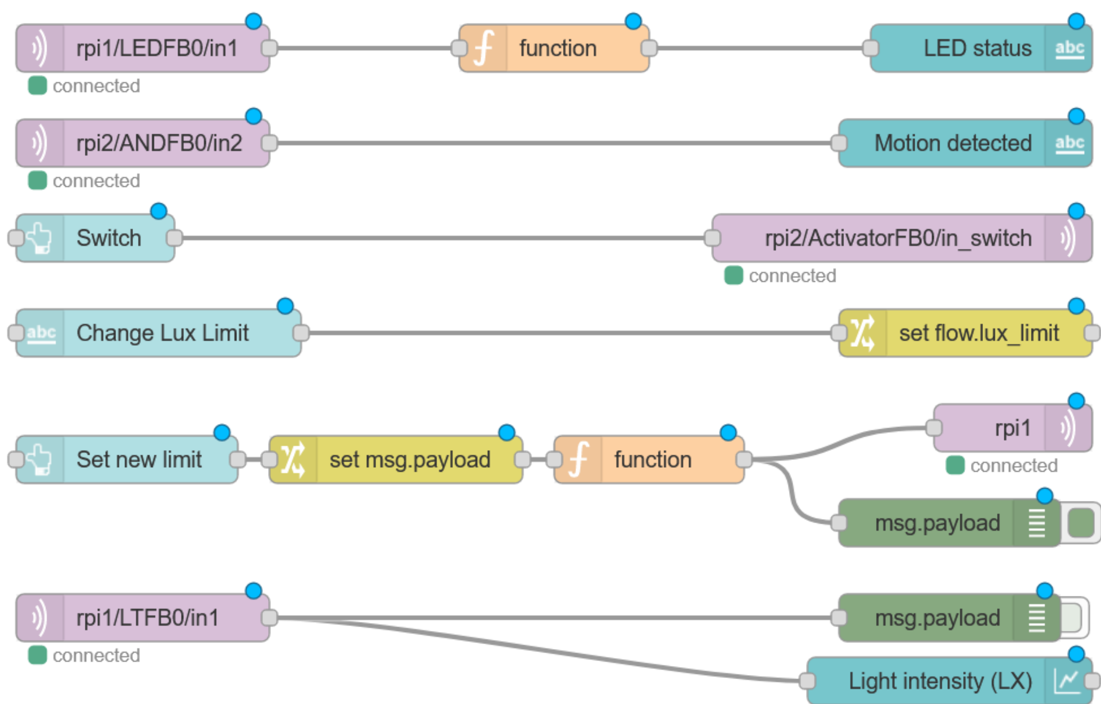


Obrázek 6.9: Uživatelské rozhraní demo aplikace vytvoření nástrojem Node-RED dashboard.

programování založené na toku dat, které spojuje dohromady více hardwarových zařízení, API a online služby. Node-RED umožňuje mimo jiné vytvářet prvky komunikující prostřednictvím protokolu MQTT, a tudíž je jej možné propojit s mým distribuovaným řídicím systémem.

Uživatelské rozhraní v Node-RED

Pro účely uživatelského HMI jsem využil existujícího řešení, které nabízí software Node-RED, konkrétně jeho rozšíření `dashboard`. V tomto nástroji jsem vytvořil jednoduché uživatelské rozhraní demo aplikace, které umožňuje v reálném čase sledovat vybrané hodnoty, jako je stav LED diody, senzorů atd. Je zde možné měnit hodnotu proměnné `LuxLimit`, která slouží jako hranice při aktivaci LED diody. Tímto způsobem je možné přizpůsobovat aplikaci různým místnostem s různým okolním osvětlením. Podoba uživatelského rozhraní je znázorněna na obrázku 6.9. Obrázek 6.10 ukazuje definici rozhraní pomocí komponent Node-RED, které znázorňují tok dat.



Obrázek 6.10: Definice rozhraní pomocí toku dat technologie Node-RED.

Kapitola 7

Závěr

Cílem této práce bylo navrhnout a vytvořit distribuovaný řídicí systém s dynamicky modifikovatelnými uzly, kterými jsou zařízení Raspberry Pi Zero. Zaměřil jsem se konkrétně na operační systém uzlů, který implementuje zázemí pro aplikace běžící v rámci distribuovaného systému. Tímto operačním systémem je runtime platforma nadefinovaná v jazyce Python 3. Uživatelskou aplikaci tvoří hierarchické uspořádání funkčních bloků, které implementují jednotlivé dílčí operace aplikace. Komunikační systém je postaven na dvou typech zpráv, servisních a aplikačních, které se publikují prostřednictvím MQTT protokolu.

Navržený systém jsem realizoval na ukázkové aplikaci, která řeší automatické osvětlení místnosti. Tato aplikace demonstruje funkcionalitu vytvořeného systému a také způsob, jakým je možné uživatelské aplikace vytvářet. Aplikace se skládá jak z atomických, tak z kompozitních funkčních bloků, které komunikují z reálnými senzory, jež jsou připojené na zařízeních Raspberry Pi Zero. Systém je vhodný převážně pro řešení týkající se chytrých domácností nebo k prototypování větších systémů.

Vytvořený systém podporuje definici uživatelské aplikace v jazyce ST, který je možné generovat například programem OpenPLC Editor. Obecně však lze aplikaci definovat a následně dynamicky rekonfigurovat sekvencí příslušných servisních zpráv. Pro zpracování definice aplikace v jazyce ST jsem vytvořil skript, který automaticky generuje potřebné servisní zprávy a publikuje je na dané uzly.

Za účelem správy systému jsem vytvořil servisní uživatelské rozhraní – HMI. Díky tomuto rozhraní je možné vzdáleně sledovat dění v systému, včetně monitorování přenášených dat v reálném čase. Pro cílové uživatele systému je však vhodnější jednodušší rozhraní. Zde můj systém poporuje napojení na již existující rozhraní, jako je například Node-RED, ale obecně jakýkoliv software, který dokáže monitorovat MQTT komunikaci.

Systém je otevřený dalšímu vývoji, ať už v oblasti optimalizací či rozšiřování jeho funkcionality. Za zmínku určitě stojí například vytvoření knihovny standardních funkčních bloků, jež by vedla k efektivnějšímu definování uživatelských aplikací.

Práce na tomto projektu značně rozšířila mé znalosti v oblasti distribuovaných systémů. Seznámil jsem se díky ní s novým odvětvím informatiky a s novým pohledem na řešení komplexních problémů využitím právě těchto systémů, což pro mě bylo velkým přínosem.

Literatura

[1] *MQTT Version 5.0* [online]. Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. 2019-03-07 [cit. 2021-03-04]. Dostupné z: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[2] Anderson, M. *What is SCADA?* [online]. 2019-06-03 [cit. 2021-03-04]. Dostupné z: https://realpars.com/scada/.

[3] Drahovský, P. *Rekonfigurovatelný IoT uzel na bázi ESP8266/ESP32*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Janoušek, V.

[4] electricaltechnology.org. *What is Distributed Control System (DCS)?* [online]. 2016-08 [cit. 2021-03-04]. Dostupné z: https://www.electricaltechnology.org/2016/08/distributed-control-system-dcs.html.

[5] inductiveautomation.com. *What is HMI?* [online]. 2018-10-08 [cit. 2021-03-04]. Dostupné z: https://www.inductiveautomation.com/resources/article/what-is-hmi.

[6] Karl Heinz John, M. T. *IEC 61131-3: Programming Industrial Automation Systems*. 1. vyd. Springer, 2001. ISBN 3-540-67752-6.

[7] Morlan, T. *SCADA 101: Local Historian Overview* [online]. 2019-11-22 [cit. 2021-03-04]. Dostupné z: https://blog.norcalcontrols.net/local-historian-overview.

[8] plcopen.org. *Structuring Program Development with IEC 61131-3* [online]. [cit. 2021-05-08]. Dostupné z: https://plcopen.org/sites/default/files/downloads/7steps.pdf.

[9] www.eclipse.org. *IEC 61499 101* [online]. [cit. 2021-05-08]. Dostupné z: https://www.eclipse.org/4diac/en_help.php?helppage=html/before4DIAC/iec61499.html.

[10] www.hivemq.com. *Getting Started with MQTT* [online]. 2020-04-24 [cit. 2021-03-04]. Dostupné z: https://www.hivemq.com/blog/how-to-get-started-with-mqtt/.

[11] www.raspberrypi.org. *About us* [online]. [cit. 2021-03-04]. Dostupné z: https://www.raspberrypi.org/about/.

[12] www.raspberrypi.org. *Raspberry Pi Zero W* [online]. [cit. 2021-04-08]. Dostupné z: https://www.raspberrypi.org/products/raspberry-pi-zero-w/.

[13] WWW.RASPBERRYPI.ORG. *Raspberry Pi hardware* [online]. 2020-07-15 [cit. 2021-03-04]. Dostupné z:
https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md.

Příloha A

Obsah přiloženého paměťového média

```
.
├── Dokumentace
│   ├── obrazky-figures
│   │   └── ...
│   ├── xszymi00-01-kapitoly-chapters.tex
│   ├── xszymi00-20-literatura-bibliography.bib
│   ├── xszymi00-30-prilohy-appendices.tex
│   ├── xszymi00.tex
│   └── ...
├── src
│   ├── demos
│   │   ├── DemoApp
│   │   │   └── ...
│   │   └── config_demoapp.st
│   ├── HMI
│   │   ├── css
│   │   │   └── w3.css
│   │   └── index.html
│   ├── SpecialFB
│   │   └── ...
│   ├── clientMQTT.py
│   ├── clientMQTT.pyc
│   ├── config.py
│   ├── hmi.py
│   ├── init.py
│   ├── runtime.log
│   └── runtime.py
├── xszymi00.pdf
└── README.md
```

Příloha B

Další vývoj práce

Při práci na tomto projektu mě napadala spousta možných nápadů, které by vedly k rozšíření současné funkcionality systému, tak jak je definována v tomto textu. Rád bych zde zmínil nápad na knihovnu standardních funkčních bloků vycházejících z normy IEC 61131-3, která by vývojářům mohla značně usnadnit vývoj uživatelských aplikací. Knihovna by obsahovala definice standardních funkčních bloků v jazyce Python, které by nebylo třeba nahrávat na uzly systému. Mezi takové bloky by patřily hlavně standardní bloky uvedené v sekci [2.3.2].

Jako další možnost budoucího vývoje se nabízí vnímat kompozitní blok ne jako subaplikaci, ale jako blok, jež je možné, podobně jako atomické bloky, nadefinovat v jazyce Python, a tím umožnit tvorbu knihovního funkčního bloku. Tento pohled na kompozity by usnadnil jejich aktivaci, která by nemusela zahrnovat veškeré informace o své vnitřní struktuře, jelikož ty by již byly obsaženy v samotném zdrojovém kódu bloku.