VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

KOORDINACE IOT NA BÁZI MICROPYTHONU POMOCÍ NODE-RED

COORDINATION OF MIROPYTHON-BASED IOT BY MEANS OF NODE-RED

BAKALÁŘSKÁ PRÁCE
BACHELOR’S THESIS

AUTOR PRÁCE **JOSEF KOLÁŘ**
AUTHOR

VEDOUCÍ PRÁCE **doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Kolář Josef**
Program: Informační technologie
Název: **Koordinace IoT na bázi MicroPythonu pomocí Node-RED**
Coordination of MicroPython-Based IoT by Means of Node-RED
Kategorie: Operační systémy
Zadání:

1. Seznamte se s existující pokusnou implementací OS pro rekonfigurovatelný IoT uzel v jazyce MicroPython na platformě ESP32, komunikující protokolem MQTT.
2. Seznamte se s nástrojem Node-RED pro koordinaci IoT.
3. Navrhněte a realizujte prostředky, umožňující využít Node-RED pro koordinaci IoT s rekonfigurovatelnými uzly na bázi MicroPythonu.
4. 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
- Node-RED. URL: https://nodered.org/

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a část návrhu.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Janoušek Vladimír, doc. Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 30. dubna 2019

Abstrakt

Cílem této práce je vytvoření prostředků pro použití nástroje Node-RED ke koordinaci fyzických uzlů v podobě čipů ESP32 ve světě Internetu věcí. To je zajištěno pomocí vlastního rozšíření tohoto nástroje umožňující nasazování různorodých aplikací na uzly simultánně vedle sebe, navrženého protokolu ve formě kanálů MQTT a vlastního firmwaru pro tyto uzly. Navržený a realizovaný firmware v jazyce MicroPython je schopen asynchronní obsluhy jednotlivých aplikací, pro které poskytuje rozhraní pro komunikaci s nástrojem Node-RED. Funkce tohoto systému je úspěšně ověřena na základě dvou praktických scénářů užití, které prokazují možnost přímého nasazení systému do praxe v oblasti automatizace – a to i díky přiloženému instalátoru firmwaru.

Abstract

The target of this thesis is to create means for using the Node-RED tool to coordinate physical nodes in the form of ESP32 chips in the Internet of Things. This is fulfilled by created Node-RED extension, the proposed MQTT channel protocol and custom firmware for these nodes. All of this suport deploy of diverse applications to nodes simultaneously side by side. The designed and implemented MicroPython firmware is capable of asynchronous operation of individual applications which provides an interface to communicate with the Node-RED. The functionality of this system has been successfully validated on the basis of two practical usage scenarios that demonstrate the possibility of direct use deployment system into practice in automation – even with the included firmware installer.

Klíčová slova

IoT, Node-RED, ESP32, automatizace, Internet věcí, MicroPython

Keywords

IoT, Node-RED, ESP32, automation, Internet of Things, MicroPython

Citace

KOLÁŘ, Josef. *Koordinace IoT na bázi MicroPythonu pomocí Node-RED.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Vladimír Janoušek, Ph.D.

Koordinace IoT na bázi MicroPythonu pomocí Node-RED

Prohlášení

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

. . . .
Josef Kolář
13. května 2019

Poděkování

Rád bych tímto vyjádřil mé poděkování doc. Ing. Vladimíru Janouškovi, Ph.D. za odborné vedení, rady a konzultace při realizaci této práce.

Obsah

1 Úvod **2**

2 Aktuální situace v Internetu věcí **3**
2.1 Existující řešení pro IoT 5
2.2 ESP32 6
2.3 MicroPython 8
2.4 Message Queuing Telemetry Transport (MQTT) 9

3 Nástroj Node-RED **13**
3.1 Základní principy nástroje Node-RED 13
3.2 Rozšíření pomocí vlastních modulů 15
3.3 Node-RED Dashboard 16

4 Návrh rozšíření Node-RED a komunikačního protokolu **18**
4.1 Požadavky na protokol 18
4.2 Kanály MQTT 19

5 Rozšíření pro Node-RED **24**
5.1 Základní konfigurační blok 24
5.2 Implementace bloku pro vstupní aplikaci 28
5.3 Implementace bloku pro výstupní aplikaci 29

6 MicroPython firmware pro ESP32 **31**
6.1 Základní požadavky pro jádro firmwaru 31
6.2 Detaily z implementace jádra 31
6.3 Implementace vstupní aplikace 34
6.4 Implementace výstupní aplikace 35
6.5 Konfigurační aplikace 36

7 Provoz **38**
7.1 Nasazení a provoz systému 38
7.2 Scénáře užití 39
7.3 Vyhodnocení provozu scénářů 41

8 Závěr **43**

Literatura **44**

A Instalace firmwaru na uzel ESP32 **46**

Kapitola 1

Úvod

Internet věcí je v roce 2019 pokračujícím fenoménem nejen pro odborníky z oblasti automatizace či vestavěných systémů. Díky dostupným prostředkům ve formě mikrokontrolérů, *System on Chip* řešeních či komunikačních modulů není v dnešní době problém realizovat monitorování nebo automatizaci domácnosti, kanceláře nebo podniku.

Experti odhadují, že do roku 2020 bude do sítě Internet připojeno až **31 miliard IoT** zařízení [15, 3], za dalších pět let poté víc jak dvojnásobný počet. Je tedy na místě se ptát na otázky dostupnosti programového vybavení pro konkrétní IoT uzly, jejich koordinaci a sběr a agregaci dat ze sítí. Vzhledem k rozšíření Internetu věcí mezi šikorou odbornou, či alespoň laickou, veřejnost je nutné také brát v potaz užiatelskou přívětivost těchot nástrojů.

V následujících kapitolách bude představen návrh a realizace komplexního rozšíření nástroje Node-RED použitého pro koordinaci uzlů sítě IoT. Jako uzly poslouží zařízení založená na čipech typu ESP32, pro které bude navržen a implementován operační systém a protokol pro komunikaci. Cílem práce je vytvoření **platformy** pro koordinaci IoT uzlů nad nástrojem Node-RED zahrnující programové rozšíření tohoto nástroje a firmware pro uzly ve formě čipů ESP32. Cílem práce naopak není detailní rozbor hardwarového řešení pro uzly sítě Internetu věcí.

Kapitola 2

Aktuální situace v Internetu věcí

Internet věcí lze z odborného hlediska klasifikovat jako síť fyzických zařízení komunikujících mezi sebou. Primárním obsahem zpráv jsou hodnoty získané v koncových zařízeních a příkazy pro jejich interakci se světem. Data pro síť získává zařízení z okolního prostředí pomocí senzorů (nejčastěji neelektrických) veličin, přímé uživatelské vstupy pomocí mechanických ovládacích prvků (tlačítka, přepínače, spínače), výstupy systému mohou být realizovány pomocí zobrazovacích jednotek, signalizačních světel či reproduktorů, stejně tak pomocí akčních členů, jako jsou motory, elektrická relé či krokové motory.

Existuje několik hlavních motivací pro zavedení Internetu věcí. Tou první je možnost globální automatizace procesů okolo nás – chytré domácnosti či chytré podniky. Díky snadno získatelným datům pro metriky je možné je vyhodnocovat v reálném čase a rozhodovat jejich na základě bez přítomnosti interakce člověka.

Systémy pro Internet věcí lze v zásadě rozdělit dle vnitřní struktury na centralizované a decentralizované – sítě centralizovaných systémů obsahují centrální prvek řídící okolní zařízení a nejčastěji i distribuující data mezi nimi. Decentralizované sítě jsou poté složeny pouze z množiny rovnocenných zařízení, u kterých dochází výhradně ke komunikaci mezi samotnými uzly bez dodatečného prostředníka. Porovnání základních vlastností těchto dvou konceptů lze nalézt v tabulce 2.1.

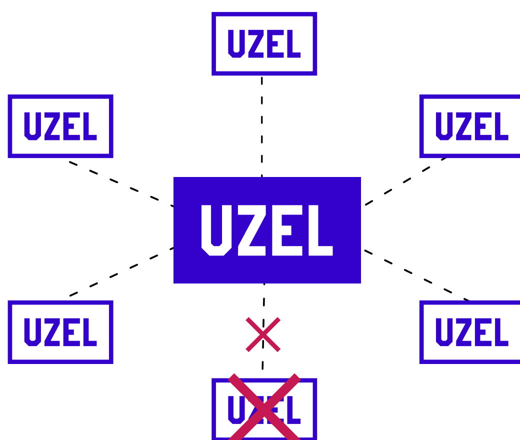
Bezpečnost systémů Internetu věcí lze rozdělit do několika úrovní [13]:

- **Fyzický uzel**
  Bezpečenost z hlediska fyzického uzlu se týká možnosti přímého přístupu potenciálního útočníka k uzlu – z principu funkce věci má uzel přístup k síti Internetu věcí. V případě kompromitovaného uzlu má tedy útočník plný přístup k datům, které síť na uzel zasílá, či má možnost do sítě produkovat napadená data. Možnou obranou je zapouzdření komunikačního kanálu uzlu do samostatného prostoru, ze kterého nemá uzel ani nepřímý přístup k ostatním uzlům.

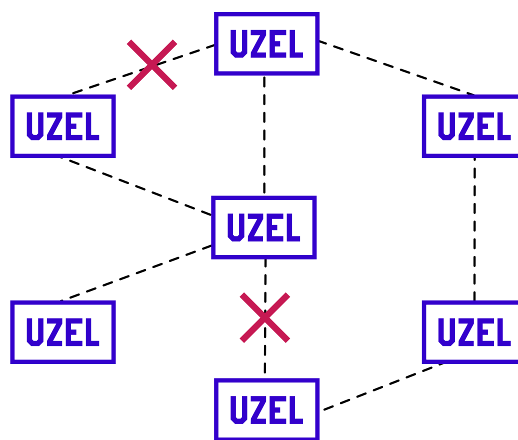
- **Komunikace**
  Od použité technologie použité ke komunikaci uzlů se odvíjí i její bezpečnost – v případě standardních bezdrátových technologií jako je WiFi, Z-Wave či ZigBee je meritum zabezpečení v konkrétní šifrované verzi konkrétní technologie, pokud existuje.

Tabulka 2.1: Srovnání vlastností základních koncepcí systému Internetu věcí [14].

	centralizovaný	**decentralizovaný**
vliv selhání uzlu na síť	v případě centrálního uzlu fatální	absence pouze konkrétního uzlu
bezpečnost dat sítě	závislá na bezpečnosti centrálního uzlu	v případě použití blockchainu velmi vysoká [17]
zdroj pravdy v síti	centrální uzel	bez jednotného zdroje
konfigurace uzlů	z centrálního uzlu principem „master-slave"	specifická pouze pro dotčené uzly
odolnost proti ztrátě dat	nižší vinou jednotného úložiště	vyšší v případě dat uložených v uzlech



(a) Centralizovaný systém IoT lze označit za graf typu hvězda – výpadek jakéhokoliv komunikačního spoje má za následek odpojení dotčeného uzlu od uzlu centrálního.



(b) Síť decentralizovaného systému IoT je obecným grafem – výpadek komunikačního spoje nemusí mít za následek nesouvislý graf. Případně grafové kružnice v tomto případě zvyšují odolnost vůči výpadkům.

Obrázek 2.1: Schémata pro centralizovaný a decentralizovaný systém IoT – na diagramech jsou naznačeny situace při selhání jedné či více komunikačních linek.

- **Centrální uzel**
  U centralizovaných systémů Internetu věcí je důležitým aspektem bezpečenosti zabezpečení centrálního uzlu. V kontextu této práce se jedná o zabezpečení nástroje Node-RED, a to především jeho editoru sítí ve formě webové stránky – komunikaci s ním lze zabezpečit pomocí šifrovaného protokolu HTTPS, kromě toho nabízí i základní autentizaci uživatelů na základě dvojice jméno-heslo.
- **Úložiště dat**
  V případě, že dochází k ukládání dat na centrální úložiště dat, které je připojeno do internetu, je otázka bezpečnosti srovnatelná se zabezpečením centrálního uzlu při centralizovaných systémech – jedná se o aplikační službu typu databáze, s čímž jsou spojeny požadavky (šifrovaná komunikace, autentizace, atd.).

2.1 Existující řešení pro IoT

Na realizaci systémů IoT se v dnešní době zaměřuje mnoho firem, stejně tak existuje několik variant s otevřeným kódem. Často se jedná o kombinaci obou potřebných stránek pro provoz – tedy k hardwarovému řešení je poskytnuto i řešení softwarové, ale existují i pouze jednostranné nástroje. Dále budou představeni největší hráči na poli jak z komerční sféry, tak sféry otevřeného zdrojového kódu.

2.1.1 OpenHAB – open Home Automation Bus

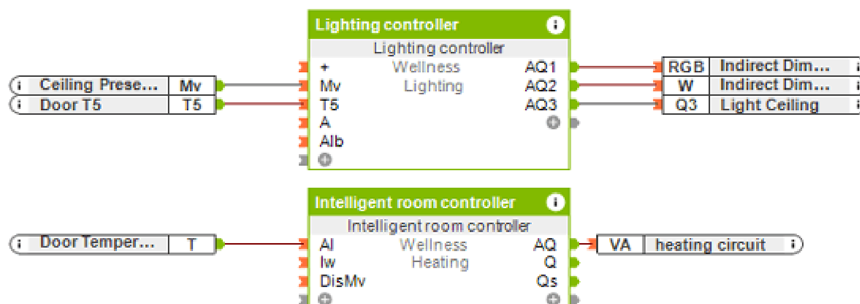
Nástroj OpenHAB je zástupcem softwarového řešení s otevřeným zdrojovým kódem v programovacím jazyce Java. Díky tomu jej lze spustit prakticky na všech dostupných hardwarových řešeních[1]. Toto řešení nabízí možnost kompletní konfigurace serveru pro řízení IoT, jednotlivé uzly sítě mohou být k serveru připojeny jak po síti Internet, tak lokálně např. pomocí rozhraní USB – k rozšíření je dostupná obsáhlá knihovna přídavných modulů. Nástroj OpenHAB je primárně založen na sestavení uživatelského rozhraní pro ovládání systému, ve kterém uživatel ovládá připojená zařízení na základě pravidel, která mohou být konfigurována staticky či dynamicky pomocí dostupného skriptování.

2.1.2 Loxone – Loxone Smart Home

Platforma Loxone je otevřený, avšak komerční systém, nabízející jak hardwarové řešení, tak odpovídající softwarové řešení – jádrem systému je miniserver s proprietární programovou výbavou, která zajišťuje obsluhu pro karty „Loxone extension“, jež jsou poté rozhraním pro komunikaci s okolním světem.

Toto řešení také nabízí experimentální funkci automatického naučení pravidel pro funkci celého systému, kdy systém detekuje své vstupy a výstupy, a na základě již existujících řešení vygeneruje pravidla. Samotná manuální konfigurace je poté možná skrz nástroj Loxone Config, který zajišťuje zaslání požadované konfigurace do miniserveru a nabízí i její lokální simulaci – dvě takováto pravidla lze vidět na obrázku 2.2.

[1] Jazyk Java pro svůj běh používá Java Virtual Machine (JVM), což je běhové prostředí pro tento jazyk, a které je schopno běhu na zařízeních založených nad architekturami *x86*, *x86_64*, *ARM* a dalšími – podpora architektury *ARM* je významná vzhledem k populárnímu mikropočítači Raspberry Pi.



Obrázek 2.2: Detail z nástroje Loxone Config – v levé části se nachází zařízení připojené do Loxone miniserveru figurující jako vstupy pro samotné pravidlo chování figurující uprostřed. Vpravo jsou poté výstupy, pomocí kterých je rozhodnutí z pravidla aplikováno zpět do reálného světa[2].

2.1.3 Domoticz

Nástroj Domoticz je založen na otevřeném zdrojovém kódu kombinující řešení jak pro fyzické uzly, tak centrální řídící prvek. Jednotlivé uzly mezi s sebou a centrálním prvkem komunikují na frekvenci pomocí několika podporovaných technologií, ať už se jedná o signál na otevřené frekvenci 433 MHz nebo standardizovaný protokol Z-Wave. Z hlediska automatizace domácností je pro tento systém zajímavá podpora internetového směrovače (serveru) Turris[3] a automatická definice pravidel na základě připojených zařízení.

2.2 ESP32

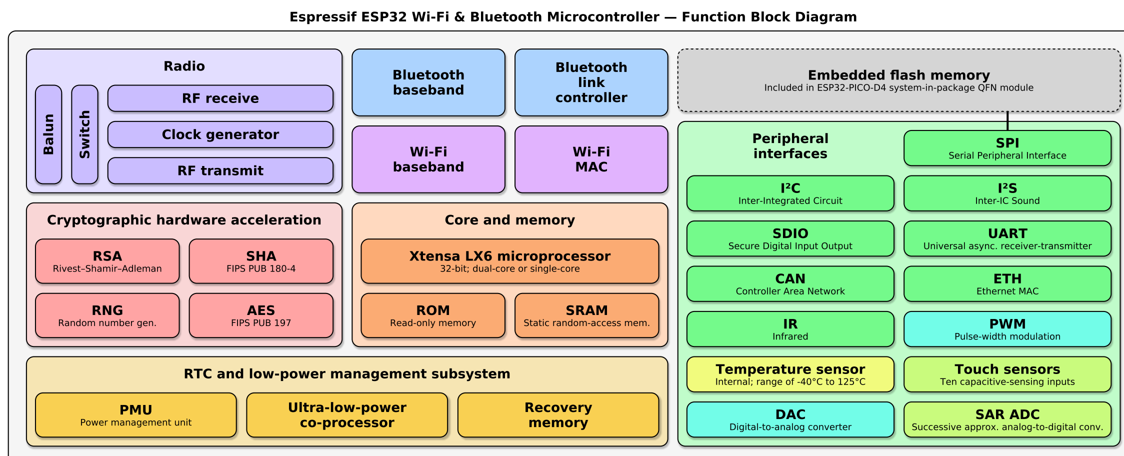
ESP32 je řada nízkonákladových SoC[4] mikrokontrolérů představená čínskou firmou Espressif Systems. Na obrázku 2.4 je zobrazen samotný čip ESP32 a varianta jeho vývojového kitu, který bude použit v této práci. Jakožto nástupce řady ESP8266 nabízí na jediném čipu ve standardní distribuci následující [8]: hl

- dvoujádrový 32bitový mikroprocesor Xtensa LX6, taktovaný na 160 či 240 MHz
- 520 KiB statické operační paměti
- WiFi ve standardu 802.11 b/g/n (včetně podpory *Long range* WiFi [1])
- Bluetooth v4.2 včetně nízkoodběrového režimu BLE
- 12 bitový ADC až s 18 kanály
- dva 2 bitové DAC
- čtyři rozhraní SPI

[2] Převzato z https://www.loxone.com/cscz/loxone-config-8.

[3] https://doc.turris.cz/gadgets/domoticz

[4] *System on Chip* je integrovaný obvod obsahující veškeré periferie (digitální, analogové a často i rádiová rozhraní) zabudované přímo v čipu. Tento princip se používá ve vestavěných systémech díky jeho nízké spotřebě.



Obrázek 2.3: Diagram jednotlivých dostupných modulů na SoC ESP32 – od jádra s procesorem a paměťmi obsahuje čip moduly pro bezdrátovou komunikaci, hardwarovou podporu kryptografie, podporu pro nízkoodběrový běh i periferie pro externí sběrnice[5].

- tři rozhraní UART
- deset GPIO pinů pro kapacitní použití
- rozhraní I^{2}C, I^{2}S, Hallovu sondu, sběrnici CAN, generátory PWM a další

Důležitou periférií je modul pro WiFi a Bluetooth komunikaci, který je obsluhován druhým jádrem procesoru. Toto chování vyžaduje podporu od běžícího operačního systému, vzhledem k tomu, že je nutné přepínat kontext procesoru právě pro správu připojení – z tohoto principu kooperace jader plyne řada bugů v čipu, které vývojáři čipu popsali v dokumentu „ECO and Workarounds for Bugs“ [7]. Pro část chyb existuje alternativní způsob, jak potlačit jejich dopad na standardní běh procesoru, část chyb byla vyřešena vydáním dalších revizí čipu ESP32.

Komunikace na WiFi protokolech využívá standardní model ISO/OSI, kdy na druhé, linkové vrstvě, modul používá MAC adresu, která je zabudována přímo v čipu při jeho výrobě[6]. Tato adresa je pro všechny vyrobené čipy ESP32 jedinečná a díky tomu, že je přístupná i pro uživatelskou programovou výbavu, je možné ji používat jako unikátní identifikátor jednotlivých čipů – této vlastnosti bude využito při návrhu protokolu dále v práci.

[5]Převzato od autora Briana Krenta – https://commons.wikimedia.org/wiki/File:Espressif_ESP32_Chip_Function_Block_Diagram.svg.

[6]MAC adresa složená ze 48 bitů je na čipu zaznamenána pomocí technologie eFuse – při výrobě dojde k nezvratnému nastavení části paměti na unikátní hodnotu.

[7]Převzato z SOS electronic s.r.o. – https://www.soselectronic.cz/products/espressif/esp32-devkitc-236729.

[8]Převzato z Digi-Key Electronics – https://www.digikey.com/product-detail/en/espressif-systems/ESP32-WROOM-32U/1904-1026-1-ND/9381735.

(a) čip ESP32 integrovaný na vývojovém kitu ESP32 Devkit C[7]

(b) samotný čip ESP32, v plechovém pouzdru je vestavěn samotný čip i taktovací krystal, vně poté anténa pro bezdrátové sítě[8]

Obrázek 2.4: Mikrokontrolér ESP32 je pro vývojové a výukové účely nejčastěji distribuován jako kompletní kit s USB konektorem pro napájení a vyvedenými piny z čipu. Pro přímé aplikace je ovšem čip dostupný i ve své surové formě.

2.3 MicroPython

MicroPython je derivát vysokoúrovňového skriptovacího programovacího jazyka Python určený pro běh na vestavěných systémech a dalších aplikacích s nízkým výpočetním výkonem. Z hlediska vestavěné knihovny nabízí téměř celou základní knihovnu z původní distribuce[9] a navíc knihovny zodpovědné za manipulaci s rozhraním mikropočítače, na kterém běží. Kromě oficiálně podporovaného sestavení pro vývojový kit „pyboard“ správci i komunitní sestavení pro vývojové kity založené na čipech „ESP2866“, „ESP32“ či „WiPy“.

Sestavení pro kity od společnosti ESP jsou založena na vývojovém frameworku ESP-IDF. Ten je zástupcem operačních systémů reálného času (RTOS), tedy systémů pro které je typické časové plánování jednotlivých úloh a kritické jejich časově přesné spuštění – to vše založené na plánovači, který je nedělitelnou součástí. Framework ESP-IDF je konkrétně postaven nad operačním systémem FreeRTOS, který je de facto standardem a největším hráčem na poli RTOS s otevřeným zdrojovým kódem [16]. Jedním z benefitů tohoto typu systému je možnost použití programového řízení komunikačních rozhraní (jako je SPI, I^2C nebo UART) – tato rozhraní lze poté implementovat, není nutný konkrétní hardwarový prvek.

I přes omezenou vestavěnou knihovnu a vysokoúrovňovost tohoto jazyka jej lze použít ve světě Internetu věcí – a to i díky jeho paměťové optimalizaci. Nespornou výhodou pro použití v Internetu věcí je i dostupnost základních knihoven pro komunikaci s okolním světem. MicroPython v základu nabízí knihovnu `machine` s třídami `Pin`, `PWM` či `ADC` – každá z těchto tříd reprezentuje způsob, jak komunikovat s okolním světem pomocí vestavěných periférií – tato část vestavěné knihovny je ovšem závislá na konkrétní distribuci interpretu jazyka MicroPython.

[9]Podporovaná podmnožina vestavěné knihovny původní jazyka je vývojáři specifikována na https://docs.micropython.org/en/latest/library/index.html#micropython-libraries.

Ústřední knihovnu pro komunikaci s okolím na vyšších vrstách poskytuje balíček `network`, který zpřístupňuje třídu `WLAN` – ta je zodpovědná za práci s modulem pro WiFi připojení a to jak v klientském módu, tak v módu přístupového bodu.

```
import machine
pin = machine.Pin(14, mode=machine.Pin.OUT)
pin.value(1) # set pin 14 to logical 1

pwm = machine.PWM(pin)
pwm.duty(512) # set DCL to near half ratio

adc = machine.ADC(0)
measured = adc.read() # read value (0-1024) from ADC
```

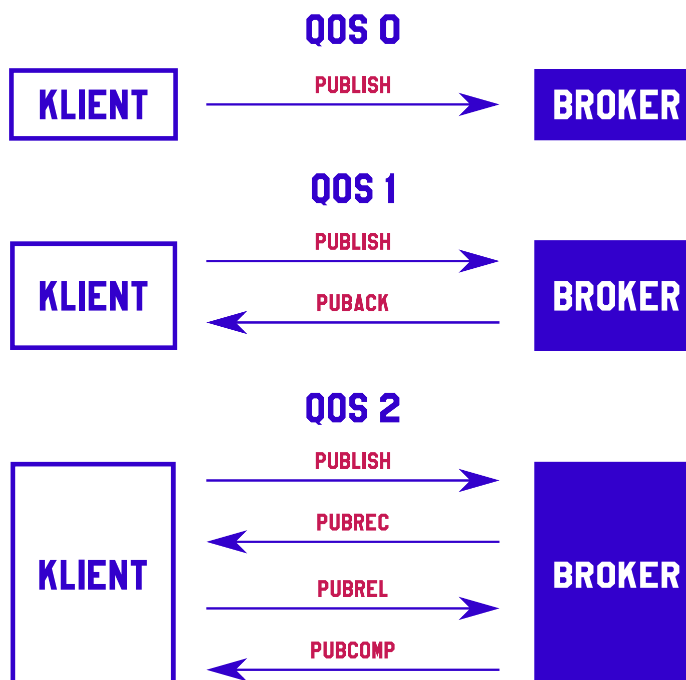
Zdrojový kód 2.1: Ukázka práce s periferiemi čipu pomocí vestavěných knihoven jazyka MicroPython – v první části se jedná o nastavení pinu na logickou hodnotu 1, prostřední část patří nastavení pulzně šířkové modulace (PWM) na pinu čipu a konec je ve znamení čtení z prvního kanálu analogově-digitálního převodníku (ADC).

2.4 Message Queuing Telemetry Transport (MQTT)

Message Queuing Telemetry Transport (MQTT) je protokol definovaný ISO standardem [9] určený pro kanálovou komunikaci zařízení mezi sebou. Jedná se o typ komunikace klient-server nad protokolem TCP/IP, server je dle svých vlastností nazýván specificky jako „broker". Klient v momentu jeho připojení vytváří sezení (*session*) na brokeru, v rámci kterého má možnost si zaregistrovat odběry jednotlivých kanálů – jejich identifikace má tvar alfanumerických řetězců oddělených pomocí znaku `/`. Benefitem přístupu s jasně definovaným oddělovačem je možnost použití zástupných znaků pro zaregistrování odběru více kanálů zároveň – této vlastnosti bude využito v pozdější části práce pro komunikaci s uzly. Znak `+` je použit pro zastoupení jedné úrovně kanálu, znak `#` pro víceúrovňové zahrnutí – demonstrace tohoto principu je shrnuta v tabulce 2.2.

Z hlediska bezpečnosti nabízí protokol možnosti na několika úrovních – tou základní a téměř nutnou je základní autentizace klienta na serveru pomocí dvojice uživatelského jména a hesla. Druhou možností je provoz protokolu MQTT nad technologií TLS/SSL, což s sebou kromě bezpečenosti přináší několik nevýhod. Tou první jsou zvýšené nároky na výkon koncových zařízení (z hlediska této práce možný problém při použití na uzlech ESP32) a kapacitu datové linky (šifrování tímto způsobem přináší datovou režii přenášených zpráv), další nevýhodou je nutnost distribuce certifikátů na obě strany komunikace. Alternativa k TLS/SSL existuje ve formě protokolu SMQTT [12], který ovšem existuje pouze ve formě návrhu a experimentální implementace, bez verze vhodné k produkčnímu nasazení.

V následujících podkapitolách budou popsány specifické vlastnosti protokolu MQTT důležité pro další postup v této práci.



Obrázek 2.5: Tři typy QoS (Quality of Service) pro MQTT - 0 pro zprávy bez potvrzování (doručení nejvýše jednou), 1 pro jednosměrné potvrzení (doručení alespoň jednou) a 2 pro obousměrné potvrzení (doručení právě jednou).

2.4.1 Příznak zprávy „retain"

Pomocí příznaku „retain" může klient u brokeru zažádat o persistenci zprávy – pro zprávu tento parametr nastaví a broker zajistí její uložení ke konkrétnímu kanálu. Dojde-li následně k zaregistrování odběru (jiným) klientem na tento kanál (včetně kanálů dle zástupných symbolů), uložená zpráva je mu automaticky ihned zaslána.

Tohoto chování lze velmi dobře využít pro kanály obsahující stav kterékoliv části ze systému – nově připojený klient pak ihned dostane zprávu o stavu a nedochází k časovému intervalu, kdy je sice klient zaregistrovaný, ale teprve čeká na novou aktualizaci stavu.

2.4.2 Quality of Service

Quality of Service (QoS) je pro MQTT vlastnost stanovující k jak důslednému potvrzování zpráv by mělo mezi brokerem a klientem docházet. Ve své podstatě stanovuje, ke kolika potvrzením odeslané zprávy musí dojít, aby byla zpráva prohlášena za úspěšně odeslanou. Na obrázku 2.5 jsou znázorněny směry zpráv protokolu mezi odesílatelem a příjemcem pro jednotlivé hodnoty dostupné pro QoS:

0 U zprávy nedochází k potvrzování, je odeslána a je na ni zapomenuto.

1 Pro zprávu je zasláno od příjemce jedno potvrzení, je tedy zajištěno alespoň jedno doručení.

2 Kromě zpětného potvrzení je potvrzeno i samotné první zpětné potvrzení, příjemce tedy odesílateli potvrdí příjetí a smazání ze svého úložiště – zpráva poté není distribuována vícenásobně.

2.4.3 Identifikace klienta „Client ID“

Možností klienta je zapsat si při připojení k brokeru své „Client ID“. Jedná se o řetězec jednoznačně identifikující konkrétního klienta – hlavním benefitem (v případě podpory persistentních sezení) je možnost zachování zaregistrovaných kanálů k odběru v případě odpojení a opětovného připojení klienta, a také uschování prozatím nedoručených zpráv s QoS nastaveným na úroveň 1 či 2.

Klient tedy zahájí komunikaci zprávou `CONNECT`, která kromě dalšího obsahuje parametry „Client ID“ a „Clean session“ – druhý zmíněný vynucuje vyčištění sezení (smazání odběru a čekajících zpráv). Broker poté odpovídá pomocí zprávy `CONNACK`, která obsahuje informaci o úspěšnosti připojení a stavu sezení (zda bylo vyčištěno či se podařilo připojit do existujícího).

2.4.4 Parametr spojení „Keep Alive“

Parametr „Keep Alive“ určuje časový interval, během kterého musí klient odpovědět na zprávu `PINGREQ` zprávou `PINGRESP` – jestliže se tak nestane, spojení je ukončeno. Toto chování je nutné kvůli nežádoucí vlastnosti protokolu TCP – v tomto protokolu, na kterém je MQTT založen, může dojít k situaci tzv. polootevřeného spojení.

Je to standardně nežádoucí stav, při kterém je jedna ze stran spojení informována o ukončení spojení (jakoukoliv vinou), druhá však ne. To vede k čekání na potvrzení odeslaných zpráv na druhé straně, která ovšem nepřichází – MQTT spojení se poté tváří jako otevřené, ale není tomu tak.

2.4.5 „Last Will“ zpráva

Zpráva s tímto příznakem je klientem nastavena při připojení k brokeru a brokerem je použita ve chvíli, kdy dojde k neočekávanému odpojení klienta. Tato zpráva „poslední vůle“ kromě samotného obsahu u sebe nese i cílový kanál, QoS či „retain“ příznak. MQTT tímto nabízí možnost klientům informovat o svém stavu odpojení autonomně bez dalšího zásahu – „Last Will“ zpráva je odeslána do daného kanálu v následujících případech:

- Klient neodpoví ve smluveném intervalu „Keep Alive“.
- Klient před ukončením spojení neodešle zprávu protokolu `DISCONNECT`.
- Broker ukončí spojení vinou chyby samotného protokolu.
- Broker detekuje IO chybu nebo chybu sítě.

Tabulka 2.2: Možnosti odběru zpráv v protokolu MQTT – protokol nabízí dva typy zástupných znaků pro zaregistrování odběru více kanálů naráz. Pomocí znaku + lze docílit zahrnutí celé jedné úrovně kanálů, znak # je poté určen k neomezenému zanoření.

kanál odběru	kanál zprávy	bude zpráva zahrnuta?
node/1	node/1	✓
	node	✗
	node/1/status	✗
	node/2	✗
node/+/status	node/1/status	✓
	node/2/status	✓
	node/foo/status	✓
	node/1/data	✗
	node	✗
	node/2	✗
node/#	node/1	✓
	node/2/status	✓
	node/2/status/data	✓
	node/	✓
	block/	✗
node/+/data/#	node/1/data/value	✓
	node/2/data/value/degrees	✓
	node/1/data/value	✓
	node/1/status/value	✗
	node/	✗
	block/	✗

Kapitola 3

Nástroj Node-RED

V této kapitole budou popsány základní principy nástroje Node-RED, vzhledem k jeho funkci ve světě Internetu věcí, a jeho důležité aspekty. Node-RED je nástroj tvůrci popsaný jako „Flow-based programming for the Internet of Things“, tedy nástroj založený na programování na datovém toku určený pro IoT. Uživateli-programátorovi nabízí jednotlivé funkční bloky (či *uzly* dle kontextu), jejichž vstupy a výstupy lze vzájemně propojovat a vytvářet tak síť jakožto celek s požadovanými funkcemi. V roce 2013 jej představila společnost IBM v rámci projektu JS Foundation pod licencí Apache 2.0. Nástroj vyžaduje běhové prostředí Node.js, tedy je implementován v programovacím jazyce Javascript, od čehož se odvíjí možnosti jeho rozšiřování.

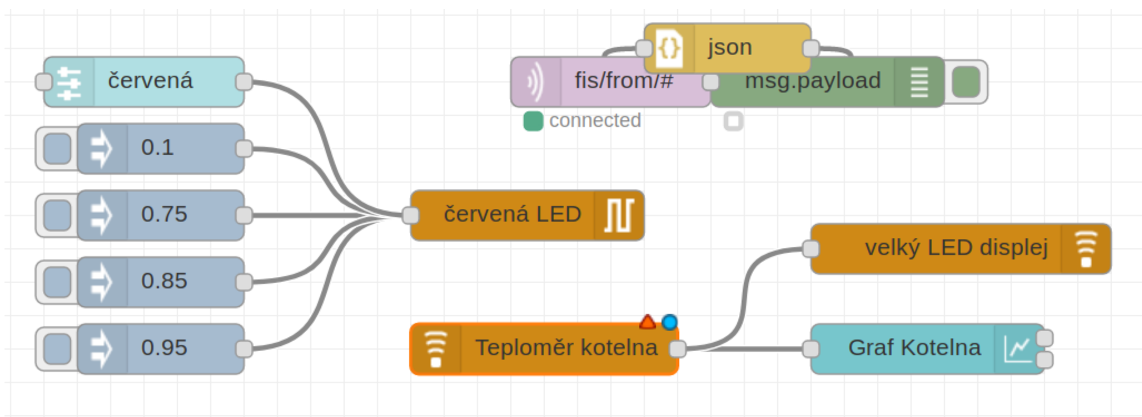
Programování datového toku

Programovací paradigma založené na editaci datového toku bylo popsáno v již v dubnu roku 1974 vědcem Dennisem. Jeho principem je programování pomocí vytváření datových spojů mezi funkčními bloky. Ve své knize „First Version of a Data Flow Procedure Language“ [4] popisuje sémantický význam funkčních bloků propojených pomocí propojů, které zajištují vzájemnou komunikaci.

3.1 Základní principy nástroje Node-RED

Vnitřní architektura nástroje Node-RED je rozdělena na dvě samostatné funkční části. Z pohledu samotného běhu je důležitější částí jádro provádějící veškeré datové operace nad samotným nadefinovaným modelem, zodpovědné za spouštění jednotlivých uživatelských uzlů, jejich synchronizaci a vzájemnou distribuci dat. Druhou částí je samotný vizuální editor, který je ve výchozím nastavení dostupný pomocí protokolu HTTP. Pomocí něj je možné uživatelsky nastavovat jednotlivé uzly a vytvářet mezi nimi datové spoje.

Toto rozdělení nabízí možnost běhu sítě mimo samotný editor, a to především kvůli bezpečnostním a výkonnostním důvodům – každé „flow“ (množina entit funkčních uzlů a jejich



Obrázek 3.1: Ukázka z nástroje Node-RED – vstupy a výstupy jednotlivých uzlů spojené pro vzájemnou komunikaci. V levé části se nachází soustava bloků typu „inject“, pomocí kterých lze manuálně odeslat zprávu do vlastního bloku ovládající periferii červené LED diody. Pravá spodní část patří měření teploty pomocí bloku teploměru, který data předává do grafu a na blok reprezentující displej.

propojení, dále jen *síť*[1]) je schopen Node-RED serializovat do formátu JSON, tedy i ukládat či načítat do, resp. ze souborů, díky čemuž lze jednotlivé sítě i snadno sdílet.

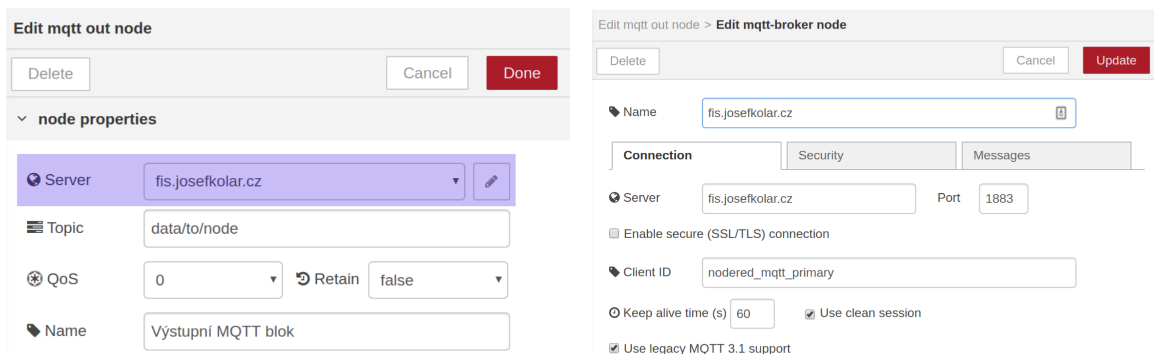
Mechanika samotných zpráv a jejich doručování je založena na několika základních premisách [10]:

1. Zprávy jsou obecně *Javascript* hodnoty, standardně datového typu `Object`, tedy množiny dvojic *klíč-hodnota*.
2. Každý z funkčních uzlů může vlastnit $0 - N$ vstupních portů, stejně tak $0 - M$ výstupních portů.
3. Jednotlivé výstupy a vstupy lze propojovat, a to i vícenásobně, tedy více výstupů lze připojit na jeden vstup, stejně tak z jednoho výstupu lze použít spojení do více vstupů.
4. Zodpovědností uzlů je provedení definované funkce na základě přijaté zprávy nebo jiného externího vstupu.
5. Zodpovědností sítě je distribuce zpráv mezi uzly a jejich asynchronní spouštění.

Node-RED bez dalších rozšíření nabízí širokou základní sadu uzlů, mezi ty nejobecněji použitelné patří:

- *function* – Blok vykonávající nakonfigurovaný uživatelský kód v programovacím jazyce Javascript pro každou příchozí zprávu.

[1] Pro „flow“ by se dal uvažovat taktéž překlad „tok“, příp. přímo „datový tok“, což ale příliš nekoresponduje s praktickou stránkou věci – „síť“ je reprezentativnější pojmenování pro strukturu vytvářenou editorem nástroje Node-RED, i například vzhledem k pojmenování Petriho sítí.



(a) konfigurace výstupního MQTT bloku – modře zvýrazněn vybraný broker MQTT

(b) konfigurace připojení k brokeru MQTT – konfigurační blok

Obrázek 3.2: Konfigurace výstupního MQTT bloku zahrnuje nastavení celého bloku (výstupní kanál, QoS, příznak „retain“ či pojmenování samotného bloku) a nastavení samotného MQTT připojení (cílový server, identifikace klienta či autentizační údaje)

- *inject* – Interaktivní blok umožnující uživateli editoru zaslat konkrétní zprávu na výstup tohoto bloku přímo z editoru (tedy internetového prohlížeče) – vhodné především pro testování či manuální zasílání zpráv.
- *debug* – Blok poskytující logovací službu, dle jeho konfigurace jsou přijaté zprávy logovány do systémové konzole či do bočního panelu v editoru – vhodné pro testování a monitorování stavu sítě.
- *switch* – Blok starající se o směrování zpráv v síti – na základně definovaných pravidel (shoda či porovnání hodnot, regulární výrazy) rozhodne, který z výstupů bude použit pro další směrování zprávy.
- *delay* – Blok umožňující časové operace nad tokem zpráv skrz tento blok - dle konfigurace zprávy buď pouze zpožďuje o konstantní hodnotu nebo omezuje průtok (šířku pásma) za časovou jednotku.
- *link* – Blok, který je schopen zprávy exportovat do jiné sítě v rámci jedné instance nástroje Node-RED – vytváří jednosměrný tunel pro mezisíťové směrování zpráv.

Pro bloky využívající externí služby nabízí Node-RED institut konfiguračních bloků. Tyto bloky jsou pouze virtuální, v síti se viditelně nevyskytují, ale je s jejich pomocí možné sdílet a znovu používat statickou konfiguraci v dalších blocích. Jedním takovým konfiguračním blokem je blok pro připojení k brokeru MQTT, jehož konfigurační formulář je vyobrazen na obrázku 3.2b)

3.2 Rozšíření pomocí vlastních modulů

Nástroj Node-RED mj. díky své otevřenosti nabízí rozšiřování pomocí vlastních bloků s vlastním chováním. Při startu tento nástroj proskenuje jmenný prostor nainstalovaných knihoven a do užšího výběru vybere knihovny s názvem začínajícím na `node-red-contrib-`

– tento prefix značí, že knihovna může obsahovat rozšiřující bloky. Standardem v ekosystému jazyka Javascript je soubor `package.json` obsahující metainformace o konkrétním balíčku (knihovně) – v tomto souboru vyhledává Node-RED klíč `node-red` a následně `nodes`.

Balíček takto exportuje uzly, které jsou zařazeny do galerie všech dostupných uzlů – reprezentaci v galerii zajišťuje soubor se zdrojovým kódem v jazyce HTML. V tomto souboru je pro uzel definován název, nápověda či ikona, ale především počet vstupů a výstupů a vlastní konfigurační parametry, které se následně zobrazují jako nastavitelné přímo v editoru – kromě standardních vstupů pro řetězce se může jednat o výběr z výčtu, výběr barvy či výběr konfiguračního bloku popsaného výše (obrázek 3.2a zachycuje možnosti definice parametrů ve výstupním bloku MQTT).

Druhým souborem je soubor v jazyce Javascript popisující chování samotného uzlu – zde jsou definovány reakce na přicházející zprávy (implementace zpracovávající příchozí zprávy), ale také připojení na služby dalších stran. Standardním obsahem tohoto souboru je funkce, která ve svém těle registruje konkrétní blok do centrálního registru bloků a následně i implementace tohoto bloku. Z editoru jsou zde dostupné nakonfigurované parametry (v parametru funkce), které mohou být následně použity pro úpravu chování bloku. Ve zdrojovém kódu 3.1 se nachází ukázka implementace jednoduchého rozšiřujícího bloku – v bloku hlavní funkce je navázána funkce obsluhující událost typu `'input'`. Na základě této události je zpracována příchozí zpráva a výsledek je opět dál odeslán jako zpráva datového typu `Object`.

```
function StringLengthNode(config) {
    RED.nodes.createNode(this, config);
    var node = this;
    node.on('input', function(msg) {
        msg.payload = msg.payload.length;
        node.send(msg);
    });
}
RED.nodes.registerType("string-length", StringLengthNode);
```

Zdrojový kód 3.1: Ukázka implementace vlastního rozšiřujícího bloku do nástroje Node-RED – uzel s jedním vstupem a výstupem, jehož funkcí je určení délky příchozí zprávy (řetězce). *Implementace pro jednoduchost neobsahuje kontrolu datových typů či práci s konfiguračními parametry uzlu.*

3.3 Node-RED Dashboard

Díky otevřenosti ekosystému nástroje Node-RED vznikl populární rozšiřující balíček s názvem `node-red-dashboard`. Tento balíček nabízí tvůrcům sítí zařadit i uživatelské rozhraní, zahrnující dynamické popisky, grafy, tlačítka a další akční a informační členy. Definice tohoto rozhraní je založena na použití standardních funkčních bloků sítě, vstupní zprávy reflektuje nástroj v uživatelském rozhraní, naopak uživatelská interakce je reflektována výstupními zprávami z bloků rozhraní. Kromě definice samotných bloků lze další vlastnosti rozhraní konfigurovat v přídavném postranním panelu v editoru – rozšíření definuje rozřazení do záložek, skupin, jejich vzájemné pozicování, pořadí a vzhled. Ukázka možností rozhraní, postaveného pomocí tohoto balíčku, je zobrazena na obrázku 3.3.

Centrální řídící panel

ZAPNOUT

Malý

Exportovat

Teplota

Úroveň 79

Zobrazit 28 úno 2019

Podsvícení

Aktuální hodnota 79 % naplnění

Úroveň

Kontakt

Jméno *

Test Testovací

Telefon

ODESLAT

ZRUŠIT

40

0 % 100

Obrázek 3.3: Ukázka uživatelského rozhraní postaveného pomocí balíčku `node-red-dashboard` – možnosti sahají od standardních prvků typu tlačítka, textového vstupu, formuláře či checkboxu, přes dynamické textové výstupy, nastavení barvy a posuvníky až ke komponentám typu výstupní grafické výstupní škály či výběru data z kalendáře.

Z pohledu sítě disponují bloky z tohoto balíčku standardními vlastnostmi – vstupy z uživatelského rozhraní jsou reflektovány pomocí výstupních zpráv bloků s informacemi o akci či změně vstupu, výstupní procedura z hlediska sítě je obdobná, bloky tohoto rozhraní jsou výstupními bloky sítě a balíček zajistí přenos informace ze sítě směrem do uživatelského rozhraní.

Kapitola 4

Návrh rozšíření Node-RED a komunikačního protokolu

V této kapitole bude popsán princip funkce koncových uzlů a způsob, kterým s nimi bude komunikováno. S využitím možností, které nabízí jazyk MicroPython, popsaných v kapitole 2.3, jsem se rozhodl koncové uzly používat víceúčelově – v jeden moment může na uzlu běžet více instancí aplikací. **Aplikace** je v kontextu této práce jednoúčelový program ovládající konkrétní periferii uzlu – pro svou činnost využívá rozhraní poskytnutého operačním systémem běžícím na ESP32. O přepínání kontextu se stará operační systém, stejně jako o připojení k Internetu, brokeru MQTT a o správu uzlu jako takového.

Aplikace je možné dělit na vstupní (data z reálného světa posílají pomocí zpráv do sítě) a na výstupní (data ze sítě transformují na interakci s okolním světem).

4.1 Požadavky na protokol

Základním spojovacím prvkem nástroje Node-RED a vlastních uzlů je rozhraní, pomocí kterého budou komunikovat. S využitím protokolu MQTT a jeho vlastností popíšu v této kapitole základní požadavky na komunikační protokol a následně jej navrhnu – ze základních požadavků poté budou plynout i požadavky na obě strany komunikace (uzly i Node-RED – centrální uzel).

Na základě povahy a potřeb obou komunikujících stran jsem navrhnul tyto základní požadavky:

1. **Pro zprávy je použit formát JSON**
   Důvodem je jeho jednoduchost, kvalitní podpora a textová podstata (tedy i snažší ladění). Ze strany Node-RED je tento formát implicitní – jedná se o „Javascript Object Notation“ – serializace a deserializace probíhají zcela přirozeně. Na straně uzlů pak poskytuje MicroPython kompletní podporu pro tento formát – deserializace probíhá do vestavěných typů.

2. **Samostatné kanály pro směry „do uzlu“ a „z uzlu“**
   Pro snížení datového toku a cílení zpráv pro konkrétní uzly je nutné oddělit komunikační kanály pro každý samostatný uzel – rozšíření na straně Node-RED zajistí

směrování do konkrétních kanálů dle konfigurace a uzly naopak odběr kanálů pouze příslušících danému uzlu.

3. **Samostatné kanály pro jednotlivé aplikace na uzlu**
   Jednotlivé aplikace běžící na uzlech je nutné v protokolu od sebe oddělovat – tzn. kromě rozlišení na úrovni všech uzlů musí dojít k rozlišení běžících aplikací na úrovni jednoho uzlu. Komunikaci bloků s příslušnými aplikacemi lze tedy specifikovat jako kanálový multiplex.

4. **Obsah zprávy je plně v režii aplikace**
   Protokol jako takový nevyžaduje *(kromě vlastních režijních zpráv)* konkrétní obsah zpráv – jejich schéma a datové typy obsahu jsou plně v režii komunikující dvojice aplikace na uzlu a bloku v nástroji Node-RED.

5. **Kanály pro status a sběr běhových informací z uzlu**
   Další z požadovaných kanálů je zpětný kanál statusu uzlu, kterým bude uzel oznamovat do sítě svůj stav (připojen či nepřipojen), případně další informace (využití úložiště, čekající data). Pro sběr běhových informací z uzlu bude použit samostatný kanál – zprávy (logy) v něm budou obsahovat stručný popis stavu, ke kterému na uzlu došlo – od výjimkových stavů až k čistě ladícím informacím.

6. **Kanál pro konfiguraci uzlu**
   Pro režijní komunikaci s uzlem je nutný samostatný konfigurační kanál, pomocí kterého bude uzel příjímat příkazy k zavedení aplikace, její překonfigurování či ukončení – dále pomocí něj mohou být přenášeny informace o vytížení uzlu či nastavení připojení k nástroji Node-RED, potažmo brokeru MQTT.

4.2 Kanály MQTT

Na základě všech vlastností protokolu MQTT popsaných v kapitole 2.4 jsem právě jej zvolil za spojovací prvek koncových uzlů a uzlu centrálního (nástroje Node-RED). V popisu navržených kanálů budou použity následující symboly:

- `NODE_ID` je jednoznačná textová identifikace uzlu v síti
- `APP_ID` je jednoznačná textová identifikace instance aplikace v rámci uzlu
- `#` je zástupný znak popsaný v kapitole 2.4
- `fis` **je zkratka z Fast IoT Solution – programového názvu pro tuto práci**

Na základě požadavků stanovených v kapitole 4.1 jsem navrhnul použití následujících kanálů MQTT (funkce budou popsány z pohledu centrálního uzlu):

- `fis/to/NODE_ID/app/APP_ID`
  Kanál je určen pro zasílání zpráv do konkrétní aplikace na uzlu – a navíc, všechny jeho podkanály (`fis/to/NODE_ID/app/APP_ID/#`) také míří do dané aplikace.

- `fis/from/NODE_ID/app/APP_ID`
  Tento kanál je určen pro odběr zpráv z konkrétní aplikace na uzlu – všechny jeho podkanály (`fis/from/NODE_ID/app/APP_ID/#`) také míří z dané aplikace. Z těchto podkanálů je poté jeden stanovený protokolem – je jím `log`, který sbírá běhové informace z kontextu konkrétní aplikace.

- `fis/from/NODE_ID/status`
  Slouží pro signalizaci stavu uzlu pro Node-RED – s pomocí parametru zpráv „retain" (popsaného v kapitole 2.4.1), zde dochází k zachování posledního známého stavu uzlu.

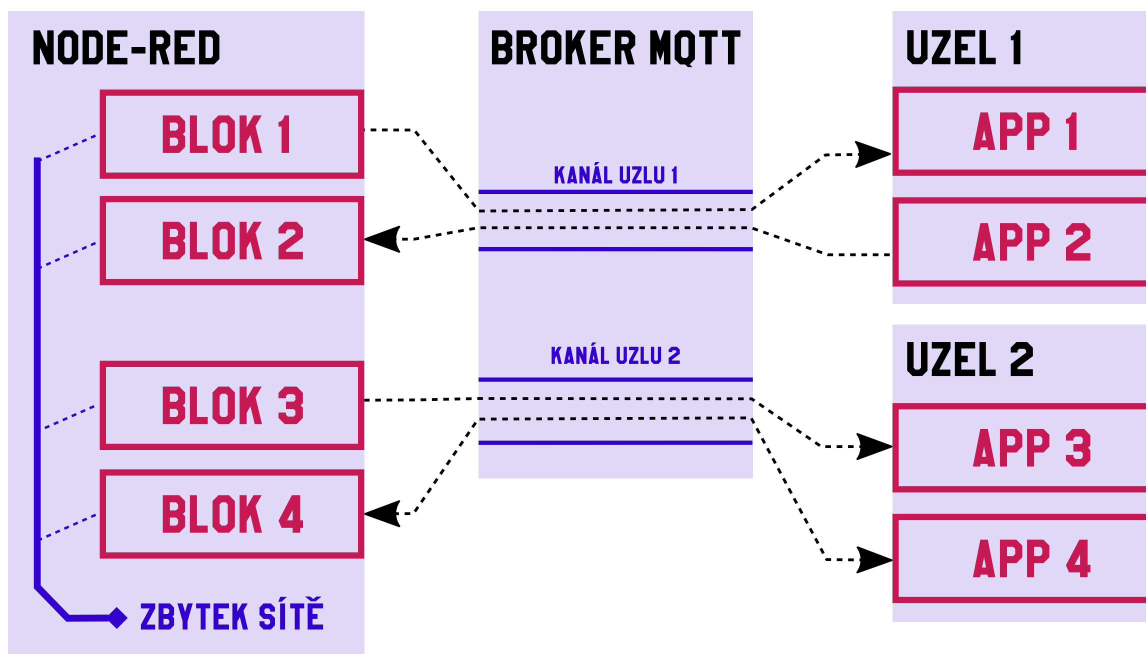
- `fis/from/NODE_ID/log`
  Slouží pro proud logů z uzlu – v kontextu celého uzlu, ne konkrétních aplikací, tedy zprávy o výjimkách na uzlu či ladících informacích.

Celkové schéma systému, jak bude z pohledu architektury a komunikace realizováno, se ocitá na obrázku 4.1 – v kapitole 5 budou popsány základní principy pro rozšíření nástroje Node-RED, kapitola 6 poté bude patřit realizaci programového vybavení pro uzly. Příklady zpráv, pomocí kterých budou tyto dvě strany komunikovat, jsou uvedeny v tabulce 4.1 – konkrétní situace, ke kterým může v rámci spojení dojít, jsou vyobrazeny na obrázku 4.2.

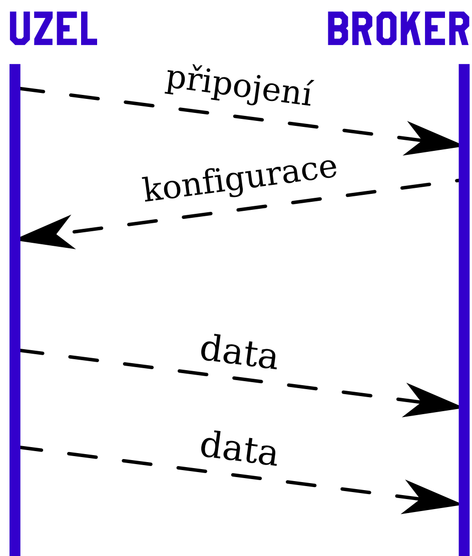
[1] Princip asynchronní smyčky bude popsán v rámci firmwaru uzlů v kapitole 6.

Tabulka 4.1: Příklady využití navrženého prokolu pro komunikaci – řádek vždy představuje jednu konkrétní zprávu v protokolu MQTT. *Pro účely tohoto přehledu jsou obsahy zpráv zkráceny.*

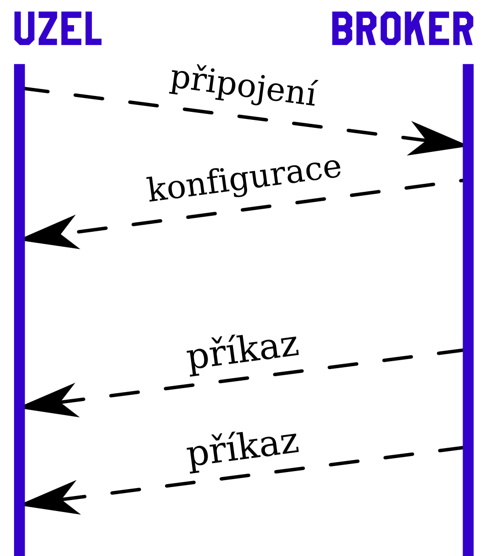
kanál	**zpráva**	**význam**
`fis/from/25a8ced/status`	`{"online": true}`	uzel s identifikátorem `25a8ced` oznamuje změnu svého stavu – své připojení
`fis/from/25a8ced/app/74cae6f`	`{"value": 0.3745}`	aplikace s identifikátorem `74cae6f` publikuje obecná data bez rozlišení subkanálu
`fis/from/25a8ced/app/74cae6f/temperature`	`{"value": 18.9}`	aplikace publikuje data do svého subkanálu `temperature`
`fis/from/25a8ced/app/74cae6f/log`	`{"msg": "invalid Pin(3)"}`	aplikace reportuje chybový stav – pin s číslem 3 není pro aplikaci vhodný/použitelný
`fis/to/25a8ced/log`	`{"msg": "42 loop tasks"}`	uzel informuje o stavu smyčky[1]
`fis/to/25a8ced/app/config`	`{"action": "reload"}`	centrální uzel zasílá zprávu pro aplikaci `config`
`fis/to/25a8ced/app/3b1cef/bottom`	`{"color": "orange"}`	centrální uzel zasílá aplikaci `3b1cef` zprávu do jejího subkanálu `bottom`



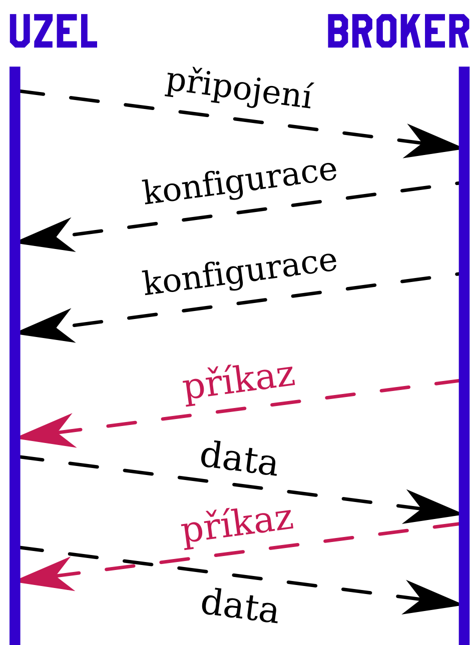
Obrázek 4.1: Schématický diagram navrženého systému – v levé části se nachází nástroj Node-RED, který obsahuje jednotlivé funkční bloky umístěné do sítě, v rámci které dochází ke směrování zpráv. Další bloky v této síti zajišťují např. uživatelské rozhraní, směrování zpráv či konkrétní funkci sítě. V prostřední části obrázku se nachází konkrétní kanály v brokeru MQTT – pomocí nich jsou bloky propojeny s odpovídajícími aplikacemi na uzlech – každý uzel je takto připojen na specifickou sadu kanálů danou jeho unikátní identifikací. Napravo jsou poté zobrazeny jednotlivé uzly se zavedenými aplikacemi – šipky značí možný směr komunikace v rámci systému, konkrétní případy jsou uvedeny v diagramu 4.2.



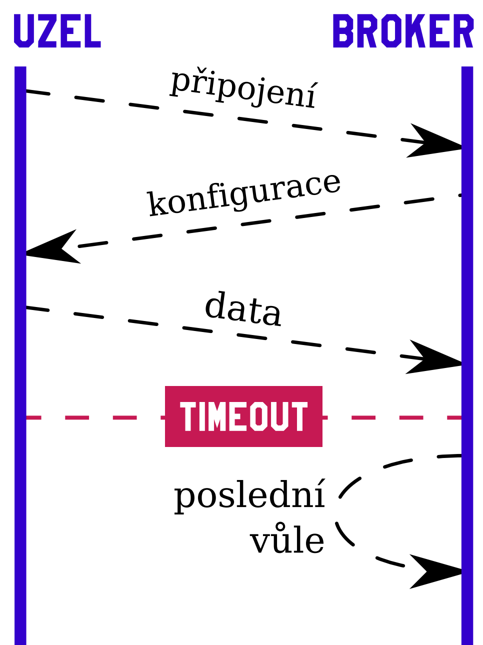
(a) Standardní situace, při které se uzel připojí, obdrží konfiguraci a zasílá bez problému data.



(b) Standardní situace, při které se uzel připojí, obdrží konfiguraci a bez problému přijímá a zpracovává příchozí zprávy.



(c) Situace s více aplikacemi na jednom uzlu – komunikace samotná není blokující, pro asynchronní jádro uzlu tedy není problém přijímat příkazy z brokeru a zároveň produkovat data zpět.



(d) Situace, při které dojde k vypršení časového limitu pro spojení „Keep Alive“, uzel je brokerem prohlášen za odpojený a do určeného kanálu odešle zprávu poslední vůle („Last Will“).

Obrázek 4.2: Diagram komunikace navrženého protokolu – zobrazeny jsou standardní posloupnosti zpráv, ke kterým na komunikační lince dochází.

Kapitola 5

Rozšíření pro Node-RED

Na základně poznatků z kapitoly 3.2 a navrženého komunikačního protokolu zde navrhnu a zrealizuji vlastní rozšíření nástroje Node-RED – předmětem budou detaily ze základní infrastruktury bloků a následně příklady konkrétních aplikací přímo nasaditelných.

5.1 Základní konfigurační blok

Základním stavebním prvkem pro další bloky je `fis-node` (reprezentovaný třídou `FisNode`) – využije se tak konfiguračních bloků, díky kterým nebude nutné přihlašovací údaje k brokeru MQTT nebo identifikacím jednotlivých uzlů zadávat více než jednou.

Jak uvádí dokumentace k nástroji Node-RED [6], nedělitelnou součástí každého bloku v Node-RED je jeho konfigurační formulář pro editor. Ten se skládá ze dvou povinných částí a jedné nepovinné – z povinných částí se jedná o programovou definici pro editor sítě v jazyce Javascript a o samotnou definici konfiguračního formuláře popsanou v jazyce HTML. Nepovinnou částí je poté uživatelská dokumentace dostupná přímo z editoru. Zkrácenou definici bloku pro editor lze vidět v ukázce 5.1, která kromě samotné registrace obsahuje dva důležité aspekty.

Prvním je registrace do kategorie `'config'`, což značí registraci konfiguračního bloku pro reprezentaci jednoho IoT uzlu (uzel tedy nemá grafickou reprezentaci v síti). Druhým aspektem je výčet parametrů pro formulář, které následně bude možné použít pro samotné chování uzlu v síti. První parametr typu `'mqtt-broker'` je určen pro konkrétní broker MQTT, pomocí kterého bude blok s uzlem spojen (jedná se o konfigurační blok poskytnutý přímo nástrojem Node-RED).

Druhým parametrem je jednoznačná identifikace bloku, jejíž formát je omezen regulárním výrazem – tento parametr vychází ze zástupného symbolu `NODE_ID` z navrženého protokolu popsaného v kapitole 4.2. Tento symbol slouží pro odlišení jednotlivých uzlů, resp. konfiguračních bloků[1]. Parametr `nodeId` tedy následně bude použit pro sestavení kanálů MQTT pro odběr a publikaci zpráv.

[1] Konkrétní tvar identifikátoru (a jeho regulárního výrazu) není pro nástroj Node-RED podstatný, k jeho upřesnění dojde až rámci firmwaru uzlů popsaného v kapitole 6.

```
RED.nodes.registerType('fis-node', {
    category: 'config',
    defaults: {
        broker: {type: "mqtt-broker", required: true, value: ""},
        nodeId: {
            value: "", required: true,
            validate: RED.validators.regex(/^[a-z0-9]{12}$/i)
        },
    },
    ...
});
```

Zdrojový kód 5.1: Registrace vlastního bloku do editoru sítě v nástroji Node-RED – kromě identifikačního klíče do registru bloků `'fis-node'` se zde nachází definice parametrů a zařazení do konfiguračních bloků pomocí zvolené kategorie `'config'`.

V ukázce 5.2 je zobrazena část třídy `FisNode`, reprezentující blok z hlediska sítě. V konstruktoru této třídy dochází k získání konfiguračních parametrů z formuláře, získání instance konfiguračního bloku pro připojení MQTT a přípravu kanálu na základě `NODE_ID`, resp. `nodeId`.

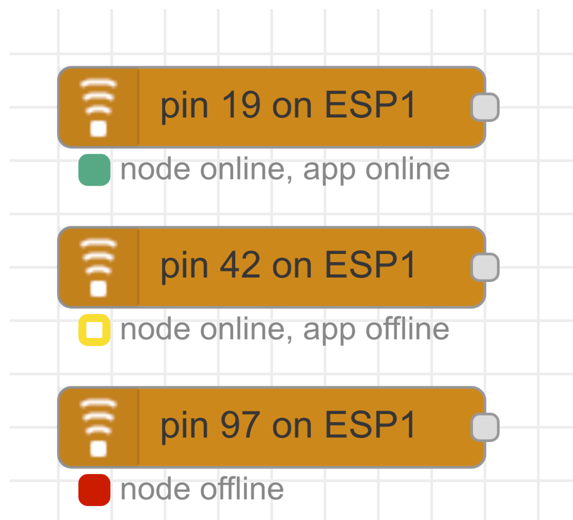
Získaná instance bloku pro MQTT je zodpovědná za správu připojení – není tedy nutné brát v potaz možné odpojení a nutnost znovupřipojení – tato instance bude následně použita k registraci odběrů a publikování zpráv do jednotlivých MQTT kanálů protokolu. Implementace tohoto bloku bohužel nemá dostupnou dokumentaci, veškeré poznatky o jeho funkci jsou tedy brány přímo z jeho zdrojového kódu[2].

```
class FisNode {
    constructor(config) {
        RED.nodes.createNode(this, config);
        this.broker = RED.nodes.getNode(config.broker);
        this.broker.connect(); // manually connect to MQTT broker
        // prepare topics for further usage
        this._publish_topic = ['fis', 'to', config.nodeId].join('/');
        this._subscribe_topic = ['fis', 'from', config.nodeId].join('/');
    }
    ...
    RED.nodes.registerType("fis-node", FisNode);
}
```

Zdrojový kód 5.2: Část konstruktoru třídy `FisNode` obsluhující připojení brokeru MQTT a přípravu kanálů pro komunikaci. Důležitá je příprava části kanálů, které jsou následně použity pro publikování a odběr zpráv – v atributu `_publish_topic` je uložen kanál s identifikací uzlu, identifikaci aplikace, dle kapitoly 4.2, připojí metoda až při publikování pro konkrétní aplikaci. Druhým předpřipraveným kanálem je kanál pro odběr zprávy z uzlu – ten je uložen v atributu `_subscribe_topic`.

Implementace třídy `FisNode` se vyznačuje několika dalšími vlastnostmi:

[2] `https://github.com/node-red/node-red/blob/master/packages/node_modules/%40node-red/nodes/core/io/10-mqtt.js`



Obrázek 5.1: Výčet stavů bloku konkrétní aplikace – třída `FisNode` vyhodnocuje na základě dostupných informací (kanál pro logy, data a status uzlu) stav odpovídajícího uzlu a aplikace, a tento stav signalizuje do editoru nástroje Node-RED. Konkrétně se jedná o stav, kdy je vše v pořádku, aplikace zasílá data (na obrázku první), stav, kdy je uzel připojen, avšak aplikace nezasílá či nepřijímá data (druhý případ) a poslední stav, kdy je uzel odpojen (třetí případ).

- Signalizace stavu do editoru – tato třída signalizuje skrz blok do editoru stav uzlu a konkrétní aplikace – jak lze vidět na obrázku 5.1, stav může nabývat tří hodnot:
  1. uzel je online, aplikace je online – vše v pořádku, u uzlu nedošlo k odpojení či vypršení limitu pro odpověď dle parametru „Keep Alive“, aplikace zasílá data
  2. uzel je online, aplikace offline – uzel je připojen, avšak aplikace loguje zprávy na úrovni chyba
  3. uzel je offline – u uzlu došlo k odpojení, buď vypnutím uzlu nebo chybou sítě (a vypršením limitu pro „Keep Alive“)
- Při smazání bloku dojde ke smazání aplikace – instance třídy `FisNode` je zaregistrovaná k odběru události typu `'close'` nad konkrétním blokem, čímž získá informaci od nástroje Node-RED, že došlo k odstranění bloku ze sítě. U tohoto procesu je využit konfigurační kanál do uzlu společně s resetováním parametru „retain“, díky čemuž nedojde k opětovnému zaslání konfigurace k již neexistujícímu bloku.
- Fronta zpráv do brokeru – třída je připravena i na stav, kdy dojde k odpojení brokeru MQTT a bloky přesto chtějí komunikovat. Implementace toto řeší pomocí navázání funkcí na událost typu `'connect'` nad objektem klienta MQTT spojení – tyto funkce ve chvíli své invokace odešlou čekající zprávu a ukončí se. Toto chování je pro bloky zcela transparentní.

Druhá povinná část, šablona konfiguračního formuláře do editoru, je úzce spjata s registrací bloku – jednotlivé parametry zde musí odpovídat formulářovým prvkům, resp. jejich názvům, jak je uvedeno v ukázce 5.3 – kvůli funkci editoru, aby byl schopen učinit formulář interaktivním a zároveň i srozumitelným pro jádro editoru a získávání hodnot z něj.

V ukázce lze také vidět speciální způsob definice HTML šablony – šablona je obalena do elementu `<script>` se specifickou hodnotou v atributu `type="text/x-red"`[3].

```
<script type="text/x-red" data-template-name="fis-node">
    <div class="form-row" id="node-config-row-broker">
        <label for="node-config-input-broker">MQTT</label>
        <input type="text" id="node-config-input-broker">
    </div>
    ...
    <div class="form-row">
        <label for="node-config-input-nodeId">Node ID</label>
        <input type="text" id="node-config-input-nodeId">
    </div>
</script>
```

Zdrojový kód 5.3: Ukázka z implementace druhé povinné části deklarace bloku – šablona formuláře v jazyce HTML obsahuje jednotlivá vstupní pole pro korespondující parametry, definované v registraci bloku do editoru v ukázce 5.2. Atribut `id="node-config-input-broker"` a odpovídající jsou důležité vzhledem k chování editoru, nutná je shoda s názvem parametru při registraci bloku – stejně jako správné spárování šablony pomocí atributu `data-template-name="fis-node"`.

Vzhledem k aplikacím (a tedy již konkrétním blokům) poskytuje základní konfigurační blok několik metod pro práci s MQTT komunikací – z těch důležitějších se jedná o `appPublish` a `appSubscribe`. Prvně jmenovaná metoda, zobrazená v ukázce 5.4, slouží k publikování zprávy pro konkrétní aplikaci (vlastnost požadovaná v kapitole 4.1) – tedy do konkrétního kanálu. Ten je v tomto místě složen z jednoznačné identifikace aplikace `appId` (z návrhu odpovídá symbolu `APP_ID`) a volitelně i podkanálu – použitá privátní metoda `_publish` poté ke kanálu před odesláním zprávy připojí identifikaci uzlu, a dojde tak k zacílení na konkrétní aplikaci na konkrétním uzlu.

```
appPublish(appId, payload, subtopic = null) {
    return this._publish(
        // subtopic is optional, filter() to avoid double slash in channel
        ['app', appId, subtopic].filter(_ => _).join('/'),
        {
            payload,
            qos: payload.qos,
            retain: payload.retain,
        }
    );
};
```

Zdrojový kód 5.4: Detail z implementace třídy `FisNode` – metoda `appPublish` poskytuje možnost konkrétnímu bloku odeslání zprávy do odpovídající aplikace na uzlu, resp. do konkrétního subkanálu.

[3] Standard HTML při tomto chování označuje element `<script>` jako „data block“ a prohlížeče jsou povinny obsah elementů s tímto atributem dále neinterpretovat.

Posledním zmíněníhodným detailem implementace třídy `FisNode` je metoda pro odběr zpráv z aplikací na uzlech `appSubscribe`. Tu mohou implementace jednotlivých bloků využít k zaregistrování odběru konkrétního kanálu, resp. konkrétní aplikace – výsledný kanál je složen ze základního kanálu (pro směr do nástroje Node-RED sestaveného již v konstruktoru), identifikace aplikace `appId` a volitelně podkanálu, opět dle navržené struktury protokolu z kapitoly 4.2. Uložená instance připojení k brokeru MQTT poté slouží k zaregistrování samotného odběru – ten je realizován pomocí funkce, kterou pro každou příchozí zprávu MQTT klient invokuje. S příchozími parametry je následně (po konverzi do formátu JSON) spuštěna funkce předaná při registraci konkrétního odběru.

```
appSubscribe(appId, callback, subtopic = null, qos = 1, ref = 0) {
    const topic = [
        // subtopic is optional, filter() to avoid double slash in channel
        this._subscribe_topic, 'app', appId, subtopic
    ].filter(_ => _).join('/');
    return this.broker.subscribe(
        topic,
        qos,
        (topic, payload) => {
            callback(topic, JSON.parse(payload));
        },
        ref,
    );
};
```

Zdrojový kód 5.5: Detail z implementace třídy `FisNode` – metoda `appSubscribe` je určená k zaregistrování odběru kanálu odpovídajícího konkrétní aplikaci na konkrétním uzlu. Parametr `qos` slouží k nastavení konkrétní hodnoty QoS pro tento odběr, `ref` je volitelná identifikace odběru, s jejíž pomocí lze mazat konkrétní odběry.

5.2 Implementace bloku pro vstupní aplikaci

Blok pro vstupní aplikaci je z pohledu nástroje Node-RED blokem, který do sítě produkuje zprávy (přijaté z třetí strany). Funkce bloku reprezentující aplikaci na uzlu vychází ze základního konfiguračního bloku, který poskytuje podporu pro komunikaci s aplikací na uzlu, její konfiguraci, správu statusu bloku a další. Konkrétní konfigurační blok (reprezentující fyzický uzel) nastaví uživatel v editoru při konfiguraci bloku a nástroj Node-RED poté instanci tohoto bloku (třídy `FisNode`) zpřístupní pomocí utilitní funkce `RED.nodes.getNode` – takto získaná instance tedy reprezentuje fyzický uzel skrz spojení MQTT. V ukázce 5.6 lze ve spodní části těla konstruktoru vidět dvě volání – nad konfiguračním blokem se prvně volá metoda `config`. Tato metoda zajišťuje napojení na servisní aplikaci určenou pro správu uzlu jako takového, a dalších aplikací – detaily její funkce budou popsány v kapitole 6.5. Volání konkrétně odešle do uzlu uživatelské nastavení společně s identifikátorem aplikace `'dht-sensor'` – zahrnut je typ senzoru a pin, na kterém je na uzlu připojen. Pro konkrétní spárování dotčeného bloku s aplikací na uzlu je odeslán i atribut bloku `this.id`, který slouží

v rámci nástroje Node-RED k unikátní identifikaci uzlu v celé síti[4], generován je náhodně implementací sítě.

Druhé volání je použití již popsané metody `appSubscribe`, blok si zde registruje odběr dat přicházejících z uzlu. Používá k tomu vlastní atribut `id`, kterým cílí na spárovanou aplikaci na uzlu, podkanál `'data'`, který následně použije i aplikace na uzlu, a funkci, pomocí které bude předávat příchozí zprávu z uzlu, resp. brokeru MQTT, do sítě nástroje Node-RED (skrz volání metody `this.send`).

```
class DhtSensor {
    constructor(config) {
        RED.nodes.createNode(this, config);
        this.fisNode = RED.nodes.getNode(config.node);

        // send app configuration to node
        this.fisNode.config('dht-sensor', this.id, {
            port: config.sensorPort,
            type: config.sensorType,
        });
        // subscribe data from node
        this.fisNode.appSubscribe(this.id, (topic, payload) => {
            this.send({
                temperature: payload.temperature,
                humidity: payload.humidity
            });
        }, 'data');
    }
}
```

Zdrojový kód 5.6: Detail implementace vstupní aplikace (z hlediska centrálního uzlu) – jedná se o aplikaci pro senzory měřící teplotu a vlhkost okolí. Konstruktor této třídy je zodpovědný za konfiguraci aplikace na uzlu za pomoci metody `config` a následnou registraci odběru zprávy z uzlu – funkce volaná na příchozí zprávy z nich přejímá data a zasílá je dále do sítě nástroje Node-RED (volání `this.send`).

5.3 Implementace bloku pro výstupní aplikaci

Blok pro výstupní aplikaci je z pohledu nástroje Node-RED blokem, který ze sítě konzumuje zprávy a zpracovává je dále. V ukázce 5.7 lze vidět implementaci bloku pro aplikaci ovládající bodový displej, jejíž detaily budou popsány v kapitole 6.4. Po standardním zavedení cílové aplikace na uzel si blok registruje odběr pomocí vlastní metody `on` – ta slouží k odběru příchozích zpráv do bloku, v tomto případě převezme příchozí zprávu a zformátuje zprávu pro aplikaci (kromě samotného textu k zobrazení podporuje volitelně i požadovanou barvu). Sestavená zpráva je odeslána pomocí `appPublish` odpovídající aplikaci do subkanálu `'text'`.

[4] Atribut `id` instancí bloků je odsledován z interní implementace běhového prostředí Node-RED – `https://github.com/node-red/node-red/blob/ed2a45e97551d9e43f079d69be8a490574e98559/packages/node_modules/%40node-red/runtime/lib/nodes/flows/Subflow.js`

```
class NeoPixelDisplay {
    constructor(config) {
        RED.nodes.createNode(this, config);
        this.fisNode = RED.nodes.getNode(config.node);
        // send app configuration to node
        this.fisNode.config('neopixel-display', this.id, {
            port: config.displayPort,
            width: config.width,
            height: config.height,
        });
        // send command to app on
        this.on('input', msg => {
            let payload = {text: msg.payload};

            if (msg.color)
                payload.color = msg.color;
            payload.retain = true;

            this.fisNode.appPublish(this.id, payload, 'text');
        });
    }
}
```

Zdrojový kód 5.7: Implementace bloku pro aplikaci ovládající bodový displej – kromě samotné konfigurace na cílovém uzlu si uzel zaregistruje funkci pro odběr události typu 'input'. Událost tohoto typu notifikuje blok o příchozí zprávě, která je v tomto případě odeslána do aplikace k zobrazení na displeji.

Použití v dalších rozšířeních

Princip konfiguračních bloků reprezentující jednotlivé fyzické uzly je dobrým vstupním bodem pro další rozšíření – ta lze realizovat pouze pomocí implementace bloku a příslušné aplikace. Základem implementace bloku by byl již existující konfigurační blok poskytující komunikační a stavové rozhraní a nutná implementace by byla pouze ta doménová, řešící konkrétní funkci bloku. Na straně uzlu, resp. firmwaru, by se jednalo o dvě části – první částí by byla implementace ovladače konkrétní periferie dle požadované funkce, druhou poté aplikace odpovídající bloku, která by měla ovladač na starost – detaily konkrétních principů firmwaru budou popsány v kapitole 6. Celé další rozšíření je možno realizovat externě k existujícímu balíčku – nástroj Node-RED registruje typy konfiguračních bloků globálně, lze je tedy používat i mimo balíčky jejich definice (stejně jako je použit blok pro broker MQTT).

Kapitola 6

MicroPython firmware pro ESP32

V této kapitole bude popsán návrh a implementace řídícího softwaru, tzv. firmwaru pro čipy ESP32 – od základního jádra zodpovědného za zavedení aplikací či připojení k brokeru MQTT až k aplikacím a jejich asynchronnímu principu.

6.1 Základní požadavky pro jádro firmwaru

Na základě navrženého protokolu, poznatků z práce P. Drahovského [5] a vlastního úsudku jsem navrhl následující požadavky na implementaci jádra firmwaru:

- Jádro je koncový bod pro komunikační linku s nástrojem Node-RED – přijímá zprávy a deleguje jejich zpracování na aplikace.
- Jádro je zodpovědné za správu připojení k internetu, potažmo brokeru MQTT, včetně přihlašovacích údajů.
- Jádro má ve správě asynchronní smyčku s požadavky aplikací.
- Jádro je zodpovědné za restartování uzlu v případě neočekávaného chybového stavu.
- Jádro hlásí stav připojení uzlu k síti a spravuje sběr logů z uzlu, které následně odesílá do příslušných kanálů.

6.2 Detaily z implementace jádra

V této kapitole budou představeny a popsány některé důležité implementační detaily jádra firmwaru. Středobodem z hlediska funkce je asynchronní smyčka událostí, jejíž implementace pochází z veřejně dostupného balíčku `micropython-async`[1]. Princip této smyčky vychází ze standardního balíku modulu `asyncio` programovacího jazyka Python a nabízí jejím uživatelům přístup ke konkurentně asynchronnímu kódu – na nejvyšší úrovni programátor

[1] https://github.com/peterhinch/micropython-async

Tabulka 6.1: Popis barevného stavového kódu uzlu – na základě chování dvou vestavěných LED diod lze odpozorovat chování uzlu a jeho stav.

červená stavová dioda	**modrá stavová dioda**	**notifikovaný stav uzlu**
nesvítí	–	uzel není připojen k napájení
svítí	nesvítí	uzel běží a čeká na zprávy či naplánované úlohy
svítí	bliká nepravidelně	uzel zpracovává zprávy či je odesílá
svítí	bliká s periodou 0,5 s	uzel se připojuje k WiFi a brokeru
svítí	svítí	chyba firmwaru

definuje pouze jednotlivé korutiny[2] (coroutines) a smyčka samotná se stará o rozdělování procesorového času a kontextu.

Vstupní bod pro zavedení jádra je metoda `Core.start`, která zařizuje jedinou věc – do inicializované smyčky zavede korutinu `Core._run`, která je zodpovědná za připojení uzlu k síti. Obě tyto metody lze vidět v ukázce 6.1, kde lze v rámci druhé zmíněné také upozornit na práci s atributem `_status_led`. V tomto atributu je uložena instance třídy `Signal`[3] nad pinem s číslem 2, na který je připojena na vývojové desce modrá LED dioda, s pomocí které uzel realizuje stavový barevný kód – tento kód jsem navrhl pro snažší testování a ladění uzlů, jeho popis se shrnut v tabulce 6.1.

[2]Korutina (coroutine) je z pohledu asynchronního přístupu k programování podpogram, jehož kód lze provádět asynchronně a zároveň má možnost jiný kód tohoto typu invokovat.

[3]Třída `Signal` je vyšší abstrakcí pro hardwarový pin GPIO než, v kapitole 2.3 zmíněná, třída `Pin` – prvně jmenovaná podporuje i piny „aktivní v 0“.

```
class Core:
    def start(self):
        self._loop.create_task(self._run())
        self._loop.run_forever()

    async def _run(self):
        self._status_led.on()
        try:
            await self._connection.connect()
        except OSError as e:
            print('Connection failed: {}.'.format(str(e)))
            return
        self._status_led.off()
    ...
```

Zdrojový kód 6.1: Metody jádra firmwaru – blokující metoda `start` je vstupní bod jádra z hlediska veřejného rozhraní, je zodpovědná za rozběhnutí smyčky událostí a naplánování běhové korutiny `_run`. Té poté stačí pouze zaktivovat klienta pro připojení k internetu, veškeré další operace obstarává klient.

Z hlediska kompozice je do běhu firmwaru zapojena i třída `MQTTConnection`. Ta, jakožto potomek třídy `MQTTClient` z balíčku `micropython-mqtt`, je zodpovědná za správu připojení k internetu a brokeru MQTT. Vlastní implementace v rámci této práce přidává do základní funkcionality třídy dvě vlastnosti. Tou první je využití ovládání stavové LED diody pro signalizaci barevného kódu a tou druhou je podpora více dvojic přihlašovacích údajů k sítím WiFi. Díky tomu lze do uzlu nakonfigurovat více známých bezdrátových sítí a uzel v případě nedostupnosti sítě či chyby ověření okamžitě zkouší připojení na další v pořadí – uzel tak v případě statických sítí není geograficky závislý. Ve chvíli úspěšného navázání komunikace a potvrzení, že připojení k brokeru MQTT je v pořádku, uzel uloží síť na první místo seznamu – při příštím startu je tak zvýšena pravděpodobnost, že dojde k rychlejšímu připojení do sítě.

Lokální správu konfiguace řeší jádro pomocí souboru `config.json`, který je uložen na systému souborů na uzlu – je tedy persistentní mezi jednotlivými běhy uzlu. Při startu je tento soubor načten a převeden z formátu JSON do slovníku, který je zpřístupněn pro použití jádru a aplikacím. Konfigurační soubor uchovává seznam přihlašovacích údajů k bezdrátovým sítím, přihlašovací údaje a adresu k brokeru MQTT.

Firmware z hlediska správy aplikací rozlišuje dva pohledy na ně – v prvním případě se jedná o aplikace dostupné k použití, reference na konkrétní třídy aplikací má uložené ve slovníku `APPS` dostupném z modulu `fis.apps`. Tento slovník používá pro klíče jednotlivé unikátní identifikátory typů aplikací, pomocí kterých lze skrz konfigurační aplikaci zavádět do uzlu konkrétní aplikace – tedy např. `'dht-sensor'` či `'neopixel-display'`. Druhým pohledem na aplikace z hlediska jádra jsou instance aplikací – již zavedené aplikace. Tyto instance jsou uloženy pod klíčem svojí unikátní identifikace ve slovníku v atributu `Core.apps` – do tohoto slovníku je při startu přímo zavedena konfigurační aplikace zodpovědná za správu

[3] Balíček `kevinkk525/micropython-mqtt` obsahuje třídu pro asynchronní práci s připojením MQTT – je dostupný skrz službu Github v repozitáři autora Kevina Köcka pod licencí MIT – https://github.com/kevinkk525/micropython-mqtt

ostatních aplikací pod statickým klíčem `'config'` – její detaily budou popsány v kapitole 6.5.

6.3 Implementace vstupní aplikace

Vstupní aplikací je z hlediska IoT sítě aplikace, která s pomocí ovladače některé z periférií získá data a dále je distribuuje do sítě pomocí protokolu MQTT. V ukázce 6.2 se nachází kód vstupní aplikace pro export dat ze senzoru DHT, který měří okolní teplotu a vlhkost a jehož ovladač je obsažen ve standardní distribuci jazyka MicroPython. Jedním z jeho specifik je neblokující chování při požadavku na změření, je tedy nutné po požadavku počkat více jak 2 sekundy, než vůbec může dojít k exportu hodnot [2]. V případě selhání komunikace se senzorem dojde ze strany ovladače k signalizaci výjimky `OSError`, na kterou aplikace reaguje zalogováním chybového stavu a vyčkáním na další interval měření. Díky použití asynchronní smyčky události není nutné v korutině `_run_measurement` při čekání na další měření používat aktivní čekání, které by blokovalo kontext procesoru pro další aplikace. Základní struktura vstupních aplikací je založena na dvou metodách:

- Korutina `init` má na starost načtení konfigurace pro aplikaci, inicializaci ovladačů a zavedení měřící korutiny do asynchronní smyčky. Měla by být schopna na základě jiné konfigurace (jiného obsahu atributu `_config`) přijmout novou konfiguraci a reinicializovat ovladače a celkové nastavení.
- Metoda `_plan_app_task` slouží k registraci aplikační smyčky do asynchronní smyčky jádra – v případě rekonfigurace je v její kompetenci resetovat již běžící aplikační korutinu a odstranit její další naplánování ve smyčce, aby mohlo dojít k opětovnému naplánování při reinicializaci aplikace.

```python
class App(BaseApp):
    _dht = _interval = None
    MEASURE_EXPORT_DELAY = 3

    async def init(self):
        self._dht = dht.DHT22(Pin(self._config.get('port')))
        self._interval = self._config.get('interval')

        self._plan_app_task(self._run_measurement())

    async def _run_measurement(self):
        while True:
            try:
                self._dht.measure()
            except OSError as e:
                await self._error(e)
                await asyncio.sleep(self._interval)
                continue

            await asyncio.sleep(self.MEASURE_EXPORT_DELAY)
            await self._publish(dict(
                temperature=self._dht.temperature(),
                humidity=self._dht.humidity(),
            ), 'data')  # custom data subtopic

            await asyncio.sleep(self._interval)
```

Zdrojový kód 6.2: Implementace vstupní aplikace pro měření teploty a vlhkosti pomocí senzoru DHT – inicializační metoda `init` načítá parametry z přijaté konfigurace a inicializuje řadič pro DHT senzor. Asynchronní měřící smyčka následně požádá ovladač o změření, počká a vyexportuje data. *Kód aplikace je pro zachování jednoduchosti zkrácen a upraven.*

6.4 Implementace výstupní aplikace

Výstupní aplikace je aplikací, které reaguje na příchozí zprávy ve svém kanálu MQTT a reaguje na ně pomocí ovladačů periférií uzlu. Příkladem uvedeným v ukázce 6.3 je výstupní aplikace řídící displej složený z LED diod technologie NeoPixel [11]. Z hlediska porovnání se vstupní aplikací se jedná o jednodušší implementaci, vzhledem k tomu, že aplikace není povinna aktivně sledovat kanál MQTT, ale činí to za ni jádro. Kromě již zmíněné korutiny `init`, ve které inicializuje ovladač k displeji se zde nachází korutina `process` zodpovědná za zpracování zprávy. Kromě parametru `payload` se samotným obsahem zprávy uloženém ve slovníku přijímá korutina i parametr obsahující seznam podkanálů, ve kterých se zpráva nachází – tyto subkanály jsou relativní vůči adresaci aplikace[4].

[4] Subkanály předané do korutiny `process` pro příchozí kanál `fis/to/NODE_ID/app/APP_ID/sub1/sub2` jsou uloženy v seznamu jako `['sub1', 'sub2']` – aplikace může použít tuto informaci k bližší specifikaci zprávy.

```python
class App(BaseApp):
    _display = _color = None

    async def init(self):
        width = int(self._config.get('width'))
        height = int(self._config.get('height'))

        self._display = NeoPixelDisplay(
            neopixel.NeoPixel(
                machine.Pin(self._config.get('port'), mode=Pin.OUT),
                width * height,  # total count of LEDs
            ),
            width, height,
        )
        self._color = self._color or 0b00100101  # low white by default

    async def process(self, payload: dict, subtopics: list):
        if payload.get('color'):
            self._color = self._display.rgb_to_color(
                *payload.get('color')
            )  # from (R, G, B)[0-255] to binary format (3b, 3b, 2b)

        if payload.get('text'):
            self._display.fill(0)  # reset display
            self._display.compact_text(
                text=payload.get('text'),
                x=payload.get('x') or 0,
                y=payload.get('y') or 0,
                color=self._color,
            )
            self._display.show()
```

Zdrojový kód 6.3: Ukázka z implementace výstupní aplikace – v korutině `init` aplikace připraví ovladač na základě parametrů doručených z editoru. Korutina `process` zodpovědná za zpracování příchozí zprávy následně tento ovladač řídí – dle nastavené barvy a textu odešle požadavek na displej.

6.5 Konfigurační aplikace

Tato aplikace je speciální režijní aplikací sloužící pro obsluhu dalších aplikací a konfiguraci celého uzlu. V registru aplikací používá statický identifikátor `'config'`, díky čemuž nemusí znát konfigurační blok `FisNode` na druhé straně komunikace konkrétní identifikaci aplikace (konkrétní `APP_ID` dle potokolu). Tato aplikace je tedy svázaná s metodou `FisNode.config`, která již byla zmíněna v rámci kapitoly 5.2. Mezi její funkce patří následující:

1. **(Re)inicializace dalších aplikací**
   Při přijetí zprávy ze strany brokeru s obsahem `{action: "init", config: ...}`

aplikace provede detekci, zda se jedná o zcela novou aplikaci zaváděnou do registru či o rekonfiguraci již existující aplikace. V obou případech provede nastavení konfiguračního slovníku do aplikace a asynchronně zažádá aplikaci o inicializaci pomocí korutiny `init`. Do registru aplikací je zavedena pod svým unikátním identifikátorem, který se nachází v použitém subkanálu konfigurační zprávy[5].

2. **Odebírání aplikací**
   Konfigurační aplikace umožňuje mazání běžících aplikací z uzlu – vyzvána je k tomu pomocí zprávy `{action: "remove"}` ve svém kanálu – díky použití subkanálu s identifiátorem aplikace není nutné v rámci obsahu zprávy přenášet aplikaci znovu. Konfigurační aplikace v tomto případě nabídne možnost aplikaci uzavřít ovladače pomocí korutiny `deinit`, odebere její případnou naplánovanou korutinu z asynchronní smyčky událostí na uzlu a následně ji z registru (a operační paměti uzlu) odstraní pomocí operátoru `del`.

3. **Změny v konfiguraci uzlu**
   Konfigurační aplikace je schopna na základě přijaté zprávy změnit aktuálně načtenou konfiguraci a příslušný soubor `config.json` – děje se tak na základě akce typu `'config'`, v těle zprávy se nachází konkrétní objekt s konfiguračními parametry. Lze tak vzdáleně do uzlu přidat podporu pro více bezdrátových sítí či změnit adresu brokeru MQTT. Při neodborné manipulaci lze ovšem takto vyřadit uzel z provozu – nakonfigurováním neznámých bezdrátových sítí apod. Tato funkce není přímo dostupná z nástroje Node-RED, pracuje s ní instalátor, jenž je popsaný v příloze A.

4. **Tvrdý reset uzlu**
   Pomocí zprávy se specifickou akcí typu `'reset'` lze uzel vzdáleně resetovat – dojde tak ke znovunačtení konfiguračního souboru, vyčištění registru aplikací a jejich následné reinicializaci. Lze tak bez přímého přístupu k uzlu vyřešit například problém nestandardního stavu v uzlu – je nutné ovšem brát v potaz, že v případě nezapnutého automatického startu firmwaru dojde po resetu k situaci, kdy je nutné firmware spustit manuálně. Tato funkce taktéž není přístupná z nástroje Node-RED, využita je instalátorem.

[5] Zprávy pro konfigurační aplikaci používají subkanál vytvořený z unikátní identifikace dotčené aplikace – tedy např. pro `fis/to/NODE_ID/app/config/42abdef87` je `'config'` identifikace konfigurační aplikace a `'42abdef87'` identifikace aplikace, o které je zaslána zpráva.

Kapitola 7

Provoz

Následující kapitola shrne možnosti realizovaného rozšíření pro koordinaci IoT, popíše, jaké je použití pro provoz sítě Internetu věcí a poté ověří realizované prostředky pro koordinaci IoT na produkčním příkladu.

7.1 Nasazení a provoz systému

Pro běh systému s realizovaným rozšířením bylo nutné zajistit následující prostředky:

- **Server**
  Pro běh požadovaných komponent celého systému je požadován server, ke kterému se budou schopny uzly pomocí bezdrátové sítě připojit a bude schopen provozovat běhové prostředí Node.js – v použitém příkladu se bude jednat o VPS s operačním systémem Debian.
- **Běhové prostředí pro nástroj Node-RED**
  Vzhledem k ekosystému nástroje Node-RED byl zvolen nástroj PM2[1], který pro aplikace v programovacím jazyce Javascript zajišťuje správu jejich běhu, tj. monitorování použitých systémových prostředků, jejich programové smyčky událostí či paralelní běh – pro správce systému poskytuje možnost Node-RED spustit.
- **Broker MQTT**
  Jako broker byla zvolena implementace s názvem *Eclipse Mosquitto*[2] – jedná se o nástroj s otevřeným zdrojovým kódem od společnosti Eclipse. Broker dle dokumentace podporuje všechny z požadovaných vlastností (parametr zprávy „retain“, tři úrovně QoS, zprávu poslední vůle atd.) a je dostupný jako aplikační balíček pro použitý operační systém.
- **Moduly ESP32**
  Pro produkční použití byly použity moduly ESP32 s deskou plošných spojů obsahující stabilizátor napětí, převodník UART-USB a lištu pinů umožňující zapojení do nepá-

[1] http://pm2.keymetrics.io/
[2] https://mosquitto.org/

jivého pole – jedná se o moduly s produktovým názvem „ESP32-DevKitC“[3] přímo od firmy Espressif Systems.

Všechny výše uvedené prostředky jsem použil pro sestavení vlastního lokálního Internetu věcí, na kterém jsem následně prováděl experimenty a ověřoval funkci realizovaných prostředků. Postup spuštění tohoto systému byl následující:

1. **Instalace firmwaru na uzel**
   Pro realizované programové vybavení pro uzly ESP32 existuje také instalátor v podobě předpisu pro nástroj `make` – instalace je složena ze tří hlavních kroků (více k tomuto postupu je uvedeno v příloze A). Prvním krokem je zavedení samotného interpretu jazyka MicroPython, následuje instalace externích knihoven jakožto závislostí realizovaného firmwaru a následně samotný kód firmwaru. Po ověření funkce firmwaru je posledním krokem zapnutí automatického zavedení firmwaru při startu – uzel je v tu chvíli připraven pro provoz čistě přes internetové připojení. Výstupem instalace je, kromě pro provoz připraveného uzlu, unikátní identifikátor uzlu, který bude následně použit pro zacílení ze strany nástroje Node-RED.

2. **Instalace Node-RED rozšíření**
   Pro instalaci vlastního rozšíření do tohoto nástroje je nutné dodržet požadavky popsané v 3.2 – realizované rozšíření všechny tyto formální požadavky splňuje, je tedy možné jej nainstalovat do jmenného prostoru knihoven nástroje Node-RED, například pomocí nástroje `npm`. Ve chvíli startu si tento nástroj knihovny poskytující bloky zaregistruje do galerie bloků, které následně nabízí uživateli.

3. **Příprava sítě**
   Dalším krokem je příprava samotné sítě v nástroji Node-RED, jedná se postupně o výběr bloků pro síť (senzory, rozhodovací prvky, výstupní prvky), jejich nastavení (použité piny na uzlu, podmínky, rozmístění prvků v GUI) včetně konfiguračních bloků (broker MQTT, identifikace uzlů získané při instalaci) a na závěr jejich propojení do sítě pomocí datových spojů. Příklad sítě, která s pomocí senzoru DHT měří teplotu a vlhkost místnosti, kterou exportuje do uživatelského rozhraní, je vyobrazen na obrázku 7.1 Proces přípravy sítě je zakončen jejím nasazením – editor zašle konfiguraci do běhové části nástroje, jenž provede reinicializaci sítě a jejích bloků.

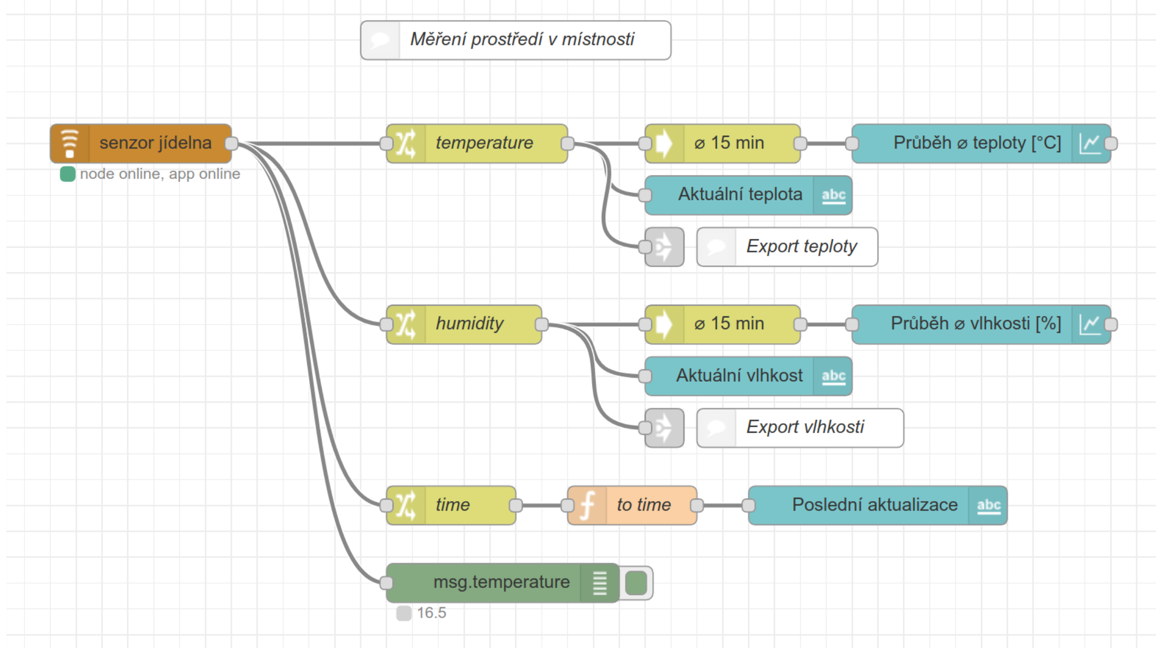
7.2 Scénáře užití

Pro ověření funkce systému byly použity dva případy užití, které budou představeny v následujících kapitolách – vždy textový popis případu užití, zobrazení implementace sítě v nástroji Node-RED a ukázka ze sestaveného uživatelského rozhraní.

7.2.1 Měření prostředí v místnosti s výstupem do uživatelského rozhraní

Síť prvního scénáře obsahuje vstupní blok reprezentující senzor typu DHT22 měřící teplotu a vlhkost okolního prostředí. Zprávy z tohoto senzoru odečítá nasazená aplikace na uzlu

[3] https://www.espressif.com/en/products/hardware/esp32-devkitc/overview



Obrázek 7.1: Implementace sítě pro scénář *Měření prostředí v místnosti s výstupem do uživatelského rozhraní* – vlevo se nachází vstupní blok reprezentující uzel s připojeným senzorem typu DHT22. Výstup tohoto bloku dále směřuje do bloků exportujících konkrétní hodnoty ze zprávy (vstupem je datový typ `Object`, výstupem `float`), ze kterých již dochází k zobrazení a vyhodnocení časového aritmetického průměru pro graf – pro snazší ladění je zde přidán blok typu `debug`, který exportuje aktuální teplotu do ladícího panelu nástroje. Pro signalizaci času poslední aktualizace slouží soustava bloků pro nastavení aktuálního času při přijetí zprávy a jeho konverze na podobu do uživatelského rozhraní.

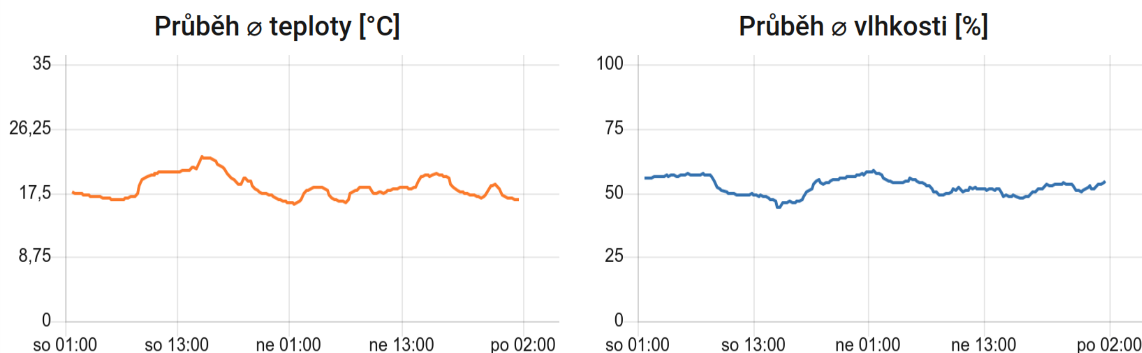
s intervalem 30 sekund a skrz firmware a broker MQTT je distribuuje do sítě nástroje Node-RED, kde jsou dále zpracovávány – dojde k vyextrahování hodnoty teploty a vlhkosti ze zprávy, zobrazení aktuální hodnoty v textovém výstupu uživatelského rozhraní a promítnutí časového aritmetického průměru hodnoty za 15 minut do grafu.

Implementace sítě pro tento scénář je zobrazena v obrázku 7.1, odpovídající část uživatelského rozhraní je poté vyobrazena na obrázku 7.2.

7.2.2 Zobrazení aktuálního času a teploty na displeji

Druhý testovací scénář obsahuje displej sestavený z LED diod technologie Neopixel, na který se zobrazuje aktuální čas a aktuální teplota v měřené místnosti z prvního scénáře – čas je vždy zobrazen po dobu X a následně je po dobu Y zobrazena teplota. Tyto hodnoty jsou uživatelem stanoveny z uživatelského rozhraní – v implementaci sítě tedy běží čítač, který na základě předaných vstupů definuje výstup do displeje. Speciálním případem je zadání statického textu do rozhraní, tento text je poté zobrazen bez vlivů čítače – ve všech případech lze použít vstup pro výběr požadované barvy displeje.

Ukázka z implementace této sítě se nachází na obrázku 7.3, sestavené uživatelské rozhraní poté na obrázku 7.4.



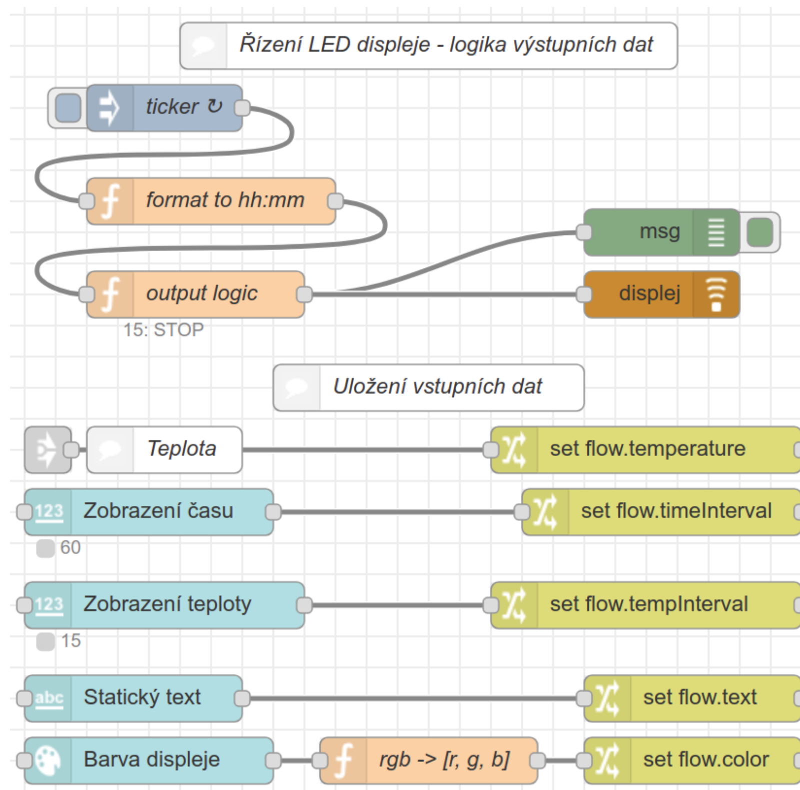
Obrázek 7.2: Ukázka z uživatelského rozhraní pro scénář *Měření prostředí v místnosti s výstupem do uživatelského rozhraní* – kromě naposledy změřených hodnot se zde nachází grafy průběhů měřených hodnot a čas poslední aktualizace.

7.3 Vyhodnocení provozu scénářů

Oba scénáře byly po dobu sedmi dnů provozovány paralelně na dvou samostatných uzlech. Během této doby se realizovanému **systému plně dařilo vykonávat zadanou funkci** – firmware uzlu se vypořádal s lokálně slabým a nestabilním signálem bezdrátové sítě pomocí procesu znovupřipojení k internetu a brokeru. Výsledkem bylo obnovené spojení, díky kterému obdržel uzel od brokeru poslední konfigurační data a data pro aplikace. Z těchto dat zreprodukoval vnitřní stav a výstupy či vstupy – zobrazený text v případě displeje či naplánované měření v případě senzoru. Toto chování bylo autorem zjištěno pomocí manuálního připojení se na konzoli uzlů přímo v místě.

V této době bylo také odpozorováno několik automatických restartů uzlů, příčin bylo postupně několik – nedostatek paměti při zavedení většího množství aplikací na uzel během testování, zvýšený výkonový odběr uzlu v případě displeje bez externího zdroje a tím i zapříčiněný pád operačního systému pro nedostatek energie v procesoru nebo stav, kdy dojde k uzavření spojení MQTT ze strany brokeru, a tím i automatickému restartu uzlu.

Oba scénáře používaly každý samostatně svůj vlastní fyzický uzel – pro navržený systém by ovšem nebyl problém provozovat oba scénáře na jednom uzlu vedle sebe, jak bylo autorem ověřeno během experimentů.



Obrázek 7.3: Implementace sítě pro scénář *Zobrazení aktuálního času a teploty na displeji* – funkce je založena na dvou částech. Spodní část je určena k ukládání aktuální stavu nastavení, tedy testického textu, intervalů zobrazení a pomocí tunelového spojení i aktuální teploty – využito je k tomu vlastnosti nástroje umožňující uložení informace v rámci jedné sítě. Horní část poté obsahuje časovač, který invokuje vlastní implementaci rozhodovací logiky, jejíž výstupem je zpráva směrovaná do blok displeje (a pro snažší ladění i do postranního panelu).

Displej

Barva displeje

Zobrazení času ⌄ **60 s** ⌃

Zobrazení teploty ⌄ **15 s** ⌃

Statický text
STOP

Obrázek 7.4: Ukázka z uživatelského rozhraní pro scénář *Měření prostředí v místnosti s výstupem do uživatelského rozhraní* – kromě naposled změřených hodnot se zde nachází grafy průběhů měřených hodnot a čas poslední aktualizace.

Kapitola 8

Závěr

Cílem této práce bylo za pomocí nástroje Node-RED navrhnout a realizovat prostředky pro koordinaci uzlů Internetu věcí založených na čipech ESP32 s programovým vybavením interpretu MicroPython. Na straně nástroje pro koordinaci Internetu věcí je možné nainstalovat rozšíření zajišťující reprezentaci fyzických uzlů v tomto nástroji – síť je poté schopna skrz toto rozšíření komunikovat s dynamicky nakonfigurovanými aplikacemi na uzlech. To je zajištěno vlastním protokolem nad kanály brokeru MQTT, ke kterému jsou nástroj Node-RED a fyzické uzly připojeny. Pro uzly na bázi čipu ESP32 je v jazyce MicroPython zrealizován firmware, jenž obsahuje asynchronní smyčku obsluhující komunikaci s brokerem a jednotlivé nakonfigurované aplikace. Důležitou vlastností takto zrealizovaného systému je jeho rozšiřitelnost – pro další aplikaci stačí pouze implementace ovladače některé z periferií čipu ESP32 a protistrana ve formě bloku do editoru nástroje Node-RED.

Pro demonstraci funkce jsou implementovány vzorové aplikace různých druhů, které jsou poté použity pro ověření funkce v rámci celé sítě – to je zajištěno pomocí testovacích scénářů popsaných v závěrečné kapitole. Z provozu těchto scénářů lze vyvodit následující závěr – realizované prostředky ve formě rozšíření a firmwaru pro fyzické uzly jsou plně připraveny pro praktické zavedení do světa Internetu věcí. Pro další použití jsou všechny zrealizované prostředky dostupné ve formě balíčků s otevřeným zdrojovým kódem skrz rozcestník na platformě GitHub[1].

[1] https://github.com/thejoeejoee/fis

Literatura

[1] Allan, A.: Long-Range WiFi for the ESP32. *Medium.com*, Únor 2018.

[2] Aosong Electronics Co., Ltd: *Digital-output relative humidity & temperature sensor/module DHT22*.

[3] Apps, N. G.: 12 IoT Trends: Experts Answering How IoT Will Evolve by 2020. [Online; navštíveno 12.04.2019].
URL https://www.newgenapps.com/blog/iot-trends-how-internet-of-things-will-evolve-by-2020

[4] Dennis, J. B.: *First Version of a Data Flow Procedure Language*. Berlin, Heidelberg: Springer-Verlag, 1974, ISBN 3-540-06859-7, 362–376 s.

[5] Drahovský, P.: *Rekonfigurovatelný IoT uzel na bázi ESP8266/ESP32*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2018.
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=20280

[6] Foundation., J.: *Node-RED Documentation*. [Online; navštíveno 04.03.2019].
URL https://nodered.org/docs/

[7] Inc., E.: *ECO and Workarounds for Bugs*. [Online; navštíveno 31.03.2019].
URL https://www.espressif.com/sites/default/files/documentation/eco_and_workarounds_for_bugs_in_esp32_en.pdf

[8] Inc., E.: *ESP32 Series Datasheet*. [Online; navštíveno 25.01.2019].
URL https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

[9] Message Queuing Telemetry Transport (MQTT) v3.1.1. Standard, International Organization for Standardization, Geneva, CH, Červen 2016.

[10] Keep, S.: Understanding Node-RED flows. *Medium.com*, Březen 2016.

[11] Kolban, N.: *Kolban's book on ESP32*. Leanpub, 2017.

[12] M. Singh and M. A. Rajan and V. L. Shivraj and P. Balamuralidhar: Secure MQTT for Internet of Things (IoT). In *2015 Fifth International Conference on Communication Systems and Network Technologies*, April 2015, s. 746–751, doi:10.1109/CSNT.2015.16.

[13] Mahmoud, R.; Yousuf, T.; Aloul, F.; aj.: Internet of things (IoT) security: Current status, challenges and prospective measures. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, Dec 2015, s. 336–341, doi:10.1109/ICITST.2015.7412116.

[14] Pihkala, H.: The Internet of Things: Why Decentralization Must Be the Next Step. *DZone*, Listopad 2018.

[15] Portal, s. T. S.: Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions). [Online; navštíveno 15.04.2019]. URL https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

[16] Shah, H.: How to select the right RTOS for your IoT needs. *SIMFORM*, Listopad 2017.

[17] Team, T. I.: A Decentralized Network for Internet of Things Powered by a Privacy-Centric Blockchain. Červenec 2018.

Příloha A

Instalace firmwaru na uzel ESP32

V repozitáři `fis-esp-firmware` se kromě samotného firmwaru pro uzly nachází i instalátor – ten je založen na předpisu konfigurace pro utilitu `make` v podobě souboru `Makefile`.

Před samotnou instalací je nutné vytvořit konfigurační soubor pro uzly `config.json`, který byl popsán v kapitole 6.2 – jako vzor pro jeho vytvoření slouží šablonový soubor `config.template.json`, který obsahuje veškeré povinné klíče pro konfiguračního hodnoty nutné pro běh uzlu.

Kompletní instalace na čistý uzel probíhá pomocí `$ make install PORT=/dev/ttyUSB0`, kde parametr `PORT` definuje virtuální terminál, na kterém je zařízení připojeno – uživatel musí disponovat právy na práci s tímto terminálem. Tento cíl postupně vykoná cíle `erase-flash`, `flash-micropython`, `install-libs` a `install-fis` – v případě selhální některé z nich lze postupovat jednotlivě.

Jakmile je nainstalován firmware, lze pomocí cíle `console` (či varianty s resetem) otevřít konzoli na uzlu a v případě, že je vše v pořádku, je v lokálním prostoru jmen dostupná proměnná `c`, jenž je referencí na firmware. Manuálním voláním `c.start()` lze firmware spustit a sledovat jeho logovací výstupy.

Pro automatické spouštění firmwaru při zapnutí je nutné volat `$ make enable-autoloader`, který na systém souborů na uzlu umístí soubor `main.py`, který má na starost automatické zavedení a reset v případě chyby. Analogicky poté `$ make disable-autoloader` vypne automatický start.

Soubor `Makefile` obsahuje celkově tyto cíle (řazeno abecedně):

`console` Otevřen REPL[1] konzoli interpretu jazyka na připojeném uzlu.

`console-with-reset` Provede tvrdý reset připojeného uzlu a následně otevře konzoli.

`disable-autoloader` Vypne automatický start firmwaru při zapnutí uzlu.

`enable-autoloader` Zapne automatický start firmwaru při zapnutí uzlu.

`erase-flash` Smaže kompletně pamět flash na připojeném uzlu.

[1] „Read Execute Print Loop“ (REPL) je postup použitý pro interaktivní vykonávání kódu interpretovaného jazyka v konzoli.

flash-micropython Nahraje stažený binární obraz interpretu jazyka MicroPython na připojený uzel.

help Vypíše nápovědu k cílům.

install-fis Nahraje firmware společně se zaváděcím souborem `boot.py` na připojený uzel – kromě toho také vypne automatické zavádění firmwaru.

install-libs Pomocí instalačního souboru provede vzdálené připojení uzlu k internetu a následně nainstaluje potřebné závislosti na uzel.

install Provede kompletní instalaci na uzel (smazání flash paměti, zavedení interpretu jazyka MicroPython, instalaci závislostí a samotného firmwaru).

put-config Na připojený uzel nahraje konfigurační soubor `config.json` z aktuálního adresáře.

put-install Na uzel nahraje instalační skript.

remote-reset Na základě předaného identikátoru uzlu `NODE_ID` provede vzdálený tvrdý reset uzlu skrz konfigurační aplikaci.

reset-chip Provede tvrdý reset na připojeném uzlu.