ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA ELEKTROTECHNICKÁ
Katedra elektroenergetiky a ekologie

Bakalářská práce

Návrh a realizace real-time komunikace pro senzorickou síť s webovou řídicí aplikací

Design and Implementation of Real-time Communication for Sensory Network with Website Based Control Application

Autor práce: Martin Zlámal
Vedoucí práce: Ing. Petr KRIST, Ph.D.

Plzeň 2015

Abstrakt

V této práci je hlavním úkolem vyřešit real-time komunikaci mezi senzorickou sítí a webovou aplikací. Toho je dosaženo použitím klasické komunikace TCP a UDP. Hlavní složkou celé práce je JavaScriptový server s databází Redis, který zajišťuje výměnu informací mezi koncovými členy celé sítě. Tyto členy jsou tvořeny mikrokontroléry od společnosti STMicroelectronics, které jsou k serveru připojeny pomocí běžných síťových prvků a Ethernetu. Výsledkem práce je funkční program, jehož funkčnost byla ověřena na praktickém zapojení sítě.

Klíčová slova

Ethernet, Sails.js, Node.js, Mikrokontrolér, Redis, RESP, TCP, UDP, Websocket, STMicroelectronics

Abstract

The main task of this thesis is to solve the real-time communication between the sensory network and the web based application. It is achieved by using standard communications TCP and UDP. The main component of this thesis is JavaScript server with Redis database which provides exchanging of information between the end members of the network. These members are formed by microcontrollers from STMicroelectronics which are connected to the server via standard network elements and Ethernet. The result is a functional program which was verified by a practical network connection.

Key Words

Ethernet, Sails.js, Node.js, Microcontroller, Redis, RESP, TCP, UDP, Websocket, STMicroelectronics

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně, s použitím odborné literatury a pramenů uvedených v seznamu, který je součástí této bakalářské práce.

Dále prohlašuji, že veškerý software, použitý při řešení této bakalářské práce, je legální.

.....
Podpis

V Plzni dne 1. 6. 2015 Martin Zlámal

Poděkování

Děkuji vedoucímu této bakalářské práce Ing. Petru Kristovi, Ph.D. za jeho čas věnovaný této práci při konzultacích a za motivaci, která mě nutila posouvat své hranice stále dál. Mé poděkování patří také společnosti UNIOSO s.r.o. za cenné připomínky při návrhu celého systému a za poskytnutí vývojových desek značky STMicroelecronics, bez kterých by vytvoření této práce nikdy nebylo možné.

Obsah

Seznam obrázků

Seznam symbolů a zkratek

ADC	Analog-to-Digital Converter
AJAX	Asynchronous JavaScript and XML
AJAJ	Asynchronous JavaScript and JSON
BGA	Ball Grid Array
BSP	Board Support Package
CAN	Controller Area Network
CPU	Central Processing Unit
CRLF	Carriage Return Line Feed
DHCP	Dynamic Host Configuration Protocol
EDA	Event-Driven Architecture
EJS	Embedded JavaScript
HAL	Hardware Abstraction Layer
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
I/O	Input/Output
IoT	Internet of Things
IP	Internet Protocol
JS	JavaScript
JSON	JavaScript Object Notation
JSONP	JSON with padding
LED	Light-Emitting Diode
MAC	Media Access Control
MVC	Model View Controller

NPM . . . . . Node Package Manager
OS . . . . . Operating System
PWM . . . . . Pulse Width Modulation
RAM . . . . . Random-Access Memory
RESP . . . . . Redis Serialization Protocol
RFC . . . . . Request for Comments
RTT . . . . . Round Trip Time
SMA . . . . . Simple Moving Average
TCP . . . . . Transmission Control Protocol
TED . . . . . Technology, Entertainment, Design
UDP . . . . . User Datagram Protocol
UTP . . . . . Unshielded Twisted Pair
UFBGA . . . . . Ultra Fine BGA
XHR . . . . . XMLHttpRequest
XML . . . . . Extensible Markup Language

1

Úvod

Cílem této práce je navrhnout komunikaci pro senzorickou síť s přihlédnutím k tomu, že by tato síť měla být ovladatelná v reálném čase z webové řídicí aplikace. Toto je velmi zásadní požadavek pro budoucí realizaci, protože z hlediska elektronických systémů je real-time komunikaci možné realizovat pomocí protokolů k tomu určených, které vymezují přenos dat do přesně definovaných časových slotů (Ethernet Powerlink, Time-triggered CAN, FlexRay). U webových aplikací žádný takový prvek neexistuje a webová řídicí aplikace se tak stává limitujícím prvkem celé sítě. Existují však metody, které se real-time komunikaci, resp. rychlé komunikaci, jak je real-time u webových aplikací všeobecně chápán, mohou velmi přiblížit. V roce 2011 bylo vydáno RFC 6455 [1], které zastřešuje nový protokol websocket, který umožňuje propojení serveru a klientské části aplikace socketem a je tak možné přenášet informace velmi vysokou rychlostí, což doposud nebylo prakticky téměř možné realizovat.

V následující části práce bude rozebrána problematika komunikace senzorické sítě s webovou řídicí aplikací, ze které vyplyne, že nejvhodnějším řešením je naprogramovat jednotlivé členy senzorické sítě co nejvíce nízkoúrovňově, následně je propojit s řídicím serverem, na kterém poběží Node.js real-time server (asynchronní single-thread) pro zpracovávání požadavků a zároveň zde poběží server pro webovou aplikaci, která bude využívat websocket protokolu coby nástroje pro komunikaci s tímto serverem. Zároveň je tato senzorická síť uváděna na příkladu administrativní budovy, resp. jakéhokoliv objektu, kde se běžně pohybují lidé a využívají konvenční elektroinstalaci, tzn. například domácí objekty, popřípadě jiné objekty podobného charakteru, kde má využití této sítě praktický přínos.

2

Real-time komunikace

Real-time komunikace představuje významný prvek v aplikacích, kde je zapotřebí přesných časových rámování přenášeného signálu. Real-time systém je buď hardwarový nebo softwarový systém, který by měl komunikovat s řídicím systémem v přesně stanovených časových periodách. Tato definice tedy neznamená, že by měl systém odpovídat nebo posílat data okamžitě, ale že garantuje reakci systému v daném časovém intervalu, a to buď reakcí na výzvu od řídicího signálu nebo ve fixní časy (tedy v relativní nebo absolutní čas) [2]. Dále lze real-time komunikaci rozdělit na tzv. soft real-time a hard real-time. Rozdíl je pouze v samotném přístupu ke spolehlivosti přenosu informace. U soft real-time přístupu je možné připustit, že se informace po nějakém čase zahodí, jelikož je již nežádoucí. Jinými slovy, pokud informace nedorazí do přijímače v určitém čase, postrádá svoji informační hodnotu. Toto by měl však být pouze ojedinělý stav. U hard real-time toto není přípustné. Tento přístup se hodí pro aplikace vyžadující velmi vysokou spolehlivost a je tak méně častý.

Obecně lze však za real-time aplikaci uvažovat systém, který reaguje na požadavky bez zbytečného dopravního zpoždění, které je například u webových aplikací naprosto běžné a odezvy v přesně definovaných časových odezvách se nepoužívají zejména z důvodu rychlosti. Je totiž mnohem důležitější odeslat data ze serveru co nejrychleji, než je posílat podle časově definovaných oken. Předejít však dopravnímu zpoždění u webových aplikací není možné. Důvod je prostý. Webová aplikace musí být dostupná pro všechny uživatele na celém světě a z toho plyne, že každý uživatel je na jiném geografickém místě a čas potřebný k dostání informace ke koncovým uživatelům není stejný. Tento problém lze částečně vyřešit distribuovaným systémem, kdy se servery přibližují uživatelům, což prakticky dělají například streamovací portály jako je YouTube. Toto řešení má svá omezení, a proto druhým způsobem, jak ušetřit čas při komunikaci s koncovým prvkem, je zjednodušit

komunikační protokol nebo se omezit na co nejméně zbytečné režie, a to i za tu cenu, že nedojde ke stoprocentnímu přenosu informace.

2.1 Hardwarové prostředky senzorické sítě

Požadavky na hardwarové prostředky této sítě nejsou v současné chvíli zadávající firmou této práce nijak definovány. Je tedy možné síť navrhnout libovolným způsobem. Vzhledem ke komplikovanosti celé problematiky bude v rámci této práce popisována síť jako striktně metalická. Veškeré informace však platí bez významnějších změn i pro bezdrátová připojení. Taková síť se tedy skládá v nejmenší konfiguraci pouze z koncového členu a serveru. S narůstajícím počtem koncových členů je zapotřebí síť patřičně rozšiřovat. Výhodou tohoto systému je fakt, že se daná síť nijak neliší od běžných metalických ethernetových sítí, tzn. že lze využít veškeré dostupné prostředky pro tvorbu této sítě a není zapotřebí vyvíjet zbytečně drahá nová zařízení.

Celá síť se tak skládá z klasického ethernetového vedení a rozbočovačů, přepínačů, popř. směrovačů. Zbývá tedy vyřešit server a koncové členy. Zde však záleží na praktické aplikaci. Vezmeme-li však v úvahu nejobyčejnější systém, server pak může být prakticky jakýkoliv počítač, který dokáže zpracovat příchozí požadavky. Tzn. musí být dostatečně výkonný a pro lepší bezpečnost celého systému také redundantní (nebo alespoň některé kritické komponenty v něm). Redundanci komponent však dobře řeší klasické servery, kde jsou redundantní například zdroj, pevné disky, řadiče a dále duální paměti, popř. procesory.

Samotné koncové prvky se pak sestávají z nízkoodběrových mikrokontrolérů, které mají menší, pro danou aplikaci však dostatečný výkon. Zde opět záleží na daném účelu koncového zařízení. Pokud má sloužit jako koncentrátor, tedy zařízení sbírající data ze senzorů, potřebuje větší výkon než například termální čidlo. Obecně však platí, že zařízení musí být dostatečně výkonná, aby bylo možné využívat bezdrátového připojení nebo Ethernetu. Potřebný výkon koncového prvku je tak dán samotným programem, který na tomto prvku poběží a dále potřebnou periferií pro připojení k serveru.

Tato síť je tedy v takovém stavu, kdy je zapojen server (nejlépe na nezávislém napájení) a senzory jsou zapojeny v ethernetové síti pomocí běžných síťových prvků. Důležité je však vyřešit, co se stane, když vypadne napájení. V tomto okamžiku síť prakticky přestane fungovat. Toto se nijak neliší od např. běžné zapojení elektroinstalace. Sice by šlo zajistit napájení koncových prvků, protože server může být zapojen na více nezávislých zdrojích elektrické energie, to však nebude např. v rodinném domě běžné. Horší případ nastane, když vypadne připojení k internetu. Zde by se nejednalo

o problém, pokud by se server nacházel v řízeném objektu. Jediný efekt by byl ten, že by nebylo možné server ovládat vzdáleně. Horší situace ovšem nastane v okamžiku, kdy je server umístěn ve vzdálené serverovně. V takovém případě je pro tuto senzorickou síť potřeba vyřešit chování v případě poruchy (tzv. disaster solution). Samotné koncové členy musí vědět, jak se chovat pokud není připojení k dispozici. To většinou není problém, protože paradoxně není potřeba řešit jejich chování. To je nutné pouze v případě zabezpečení objektů. Starostí koncových členů totiž není např. vypnout světlo, pokud není systém připojen k internetu. V takovém objektu je však zapotřebí zařadit do sítě zařízení, které bude přijímat od serveru povely a obsluhovat síť. V případě přerušení spojení se serverem převezme toto zařízení kontrolu nad sítí a uvede objekt do dočasného módu než se problém vyřeší nebo než přijede servis. Bude tak možné i nadále ovládat alespoň na základní úrovni většinu zařízení. Je však vhodné, aby co nejvíce prvků vědělo, jak se v takové situaci zachovat, pokud je to možné.

2.2 Real-time ve webových aplikacích

Ve webových aplikacích žádný real-time jako takový v podstatě neexistuje. Existují však technologie, které umožňují rychlou komunikaci s webovým serverem, resp. rychlou výměnu dat, což vždy nemusí být jedno a to samé. Je totiž rozdíl mezi tím, jestli je nutné při každém požadavku sestavovat nové spojení se serverem, nebo je možné bez další režie rovnou vyměňovat data.

Jedním z typických zástupců je AJAX (popř. AJAJ). Jedná se jednosměrný mechanismus, kdy se po periodické akci nebo například při stisku tlačítka vyvolá JavaScriptová akce, která naváže HTTP spojení se serverem a získá data v závislosti na požadavku. Následně překreslí část stránky obsahující nová data. Nedojde tak k obnovení celé stránky, ke kterému by došlo při běžném pohybu návštěvníka na stránce. Výhodou je, že není zapotřebí přenášet celou stránku. Nevýhodou však je možný nárůst HTTP požadavků na server a hlavně nutnost vyjednat se serverem spojení při každém požadavku, což je časově velmi náročné. Pro tuto aplikaci je proto použití AJAXu nevhodné.

Oproti tomu websocket [1] je protokol, který umožňuje otevřít socket mezi serverem a prohlížečem a pomocí rámců posílat obousměrně informace. Vyjednat spojení se serverem tak stačí pouze jednou při otevření webové stránky a následně je možné velmi rychle se stránkou komunikovat. Zároveň se periodicky kontroluje, jestli je stránka stále aktivní (tzv. heartbeat) a pokud ne, server spojení uzavře. Nespornou výhodou je také fakt, že websocket využívá principu event-driven, takže kromě periodické kontroly ak-

tivního spojení je možné posílat data pouze pokud je to nutné, což hodně ušetří na komunikaci mezi serverem a browserem. Websocket je nezávislý protokol založený na TCP. Jediné, kdy využívá HTTP, je v okamžiku vyjednávání spojení [1]. V ten okamžik posílá na server požadavek na změnu protokolu na websocket. Níže je uvedený HTTP požadavek na server. Pro přehlednost jsou umazány s websocketem nijak nesouvisející hlavičky (např. `Accept-Encoding`, `Accept-Language`, `Cookie`, `User-Agent` a další).

```
1 GET ws://localhost.dev:1337/socket.io/1/websocket/... HTTP/1.1
2 Host: localhost.dev:1337
3 Connection: Upgrade
4 Upgrade: websocket
5 Origin: http://localhost.dev:1337
6 Sec-WebSocket-Version: 13
7 Sec-WebSocket-Key: ehMTznt9qCyM3Iu5HC9YSA==
8 Sec-WebSocket-Extensions: permessage-deflate;
  ↪ client_max_window_bits
```

Důležitá je zejména hlavička `Connection`. Tato hlavička serveru říká, že je požadována změna protokolu z `HTTP/1.1` na nějaký jiný. Hlavička `Upgrade` pak říká na jaký. Těchto protokolů zde může být uvedeno i více. V tomto případě je však požadován pouze websocket. Na základě tohoto požadavku server odpovídá:

```
1 HTTP/1.1 101 Switching Protocols
2 Upgrade: websocket
3 Connection: Upgrade
4 Sec-WebSocket-Accept: TM+QlyuB6XGn/7ei9mnaCRepx9M=
```

Server může tuto hlavičku odmítnout a ponechat `HTTP/1.1`. Tento server však požadavku vyhověl a změnil protokol na požadovaný websocket. Dnešní běžné prohlížeče tomuto protokolu bez problémů rozumí, nicméně pro případ toho, že by webovou stránku otevřel uživatel ve starším prohlížeči, jsou většinou k dispozici doplňková řešení ve formě jiných technologií tak, aby stránka fungovala. V tomto případě se jedná zejména o XHR-polling a JSONP-polling.

2.3 TCP

Protokol TCP je jedním ze dvou transportních protokolů [3], které tento systém využívá. Stejně tak jako UDP je zde tento protokol rozebírán zejména

z toho důvodu, že právě na TCP paketech a UDP datagramech je vystavěna komunikace mezi koncentrátory a serverem. Oproti UDP protokolu se jedná o poměrně komplikovanou a tedy i časově náročnou komunikaci. Posloupnost komunikace je následující, přičemž na adrese `192.168.0.20` se nachází server:

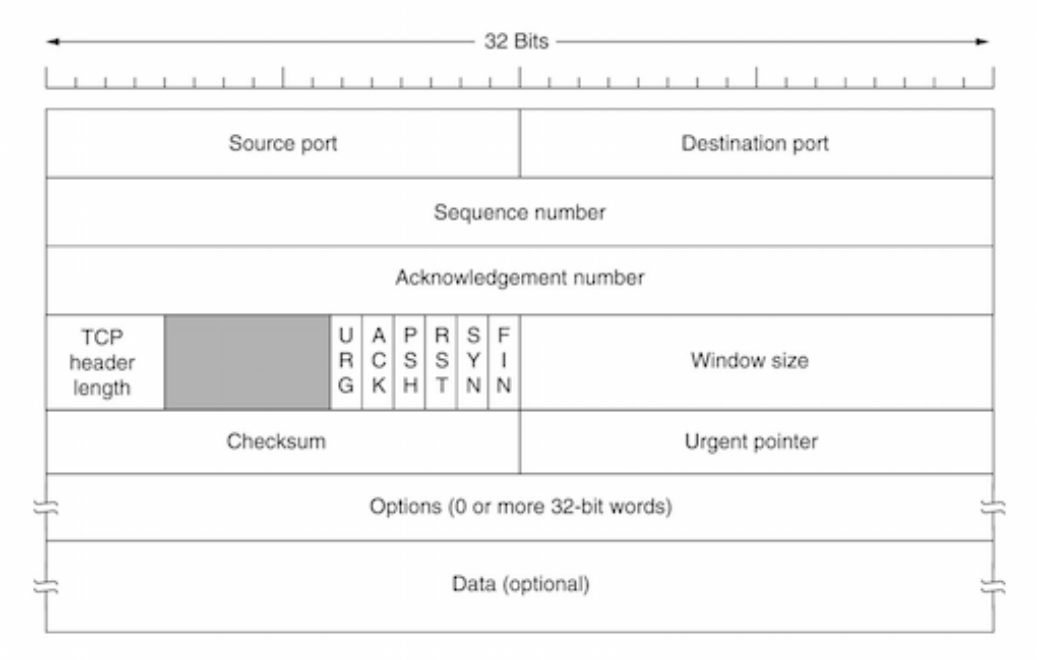
```
192.168.0.11 -> 192.168.0.20 : SYN
192.168.0.11 <- 192.168.0.20 : SYN, ACK
192.168.0.11 -> 192.168.0.20 : ACK
```

Spojení tedy probíhá zhruba následovně. Koncentrátor (`192.168.0.11`) otevírá TCP spojení vysláním požadavku na spojení příznakem `SYN`. Server potvrzuje spojení pomocí příznaku `ACK` a vysílá také požadavek na spojení (`SYN`). Koncentrátor toto spojení přijímá pomocí `ACK` příznaku, čímž je spojení ustanoveno. Tomuto procesu se říká třícestné zahájení spojení (anglicky three-way handshake) [3]. Umístění těchto příznaků v TCP paketu je znázorněno na obrázku 2.1. Samotné poslání jednoho paketu včetně dat a uzavření spojení pak vypadá následovně:

```
192.168.0.11 -> 192.168.0.20 : SYN
192.168.0.11 <- 192.168.0.20 : SYN, ACK
192.168.0.11 -> 192.168.0.20 : FIN, PSH, ACK   # přenos dat
192.168.0.11 <- 192.168.0.20 : ACK
192.168.0.11 <- 192.168.0.20 : FIN, ACK
192.168.0.11 -> 192.168.0.20 : ACK
```

Začátek spojení (třícestné zahájení) zůstává stejný, ale pro úsporu množství přenášených informací se hned při potvrzení spojení pomocí `ACK` posílají na server data (`PSH`) a zároveň se posílá požadavek na ukončení spojení (`FIN`). Následně server potvrzuje přijetí dat (`ACK`) a ihned také zasílá požadavek na ukončení spojení z jeho strany (`FIN, ACK`). Nakonec i koncentrátor potvrzuje uzavření spojení (`ACK`). Je tedy zřejmé, že i po prvním `FIN` příznaku dochází k další komunikaci. Nutně tedy tento příznak neznamená úplný konec spojení, ale pouze konec z jedné strany. V tomto případě uzavírá spojení sám koncentrátor, není to však nutné. Spojení může stejným způsobem ukončit i server. V současné chvíli nejsou mezi serverem a koncentrátorem implementovány perzistentní sockety, tedy sockety, které se neuzavírají a přetrvávají do další komunikace (obdoba websocketu).

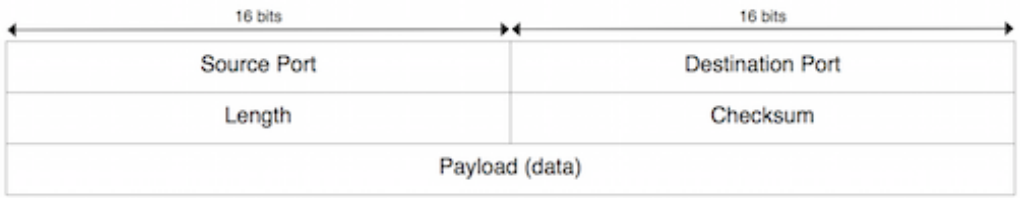
Nespornou výhodou TCP je fakt, že tento protokol zajišťuje to, že daný paket dorazí na cílovou adresu. To u UDP neplatí. TCP se proto používá pro přenos kritických informací (například vypnutí/zapnutí světla). Jsou to operace, které se musí bezpodmínečně vykonat a jejich nevykonání by vedlo pro uživatele k velmi zvláštnímu chování sítě.



Obrázek 2.1: Uspořádání TCP paketu
zdroj: `http://serverfault.com/a/8986`

2.4 UDP

UDP je oproti TCP protokol typu „fire and forget“, tedy informace se vyšle a v tu chvíli se zdrojové zařízení přestává o tuto informaci starat. Zdrojové zařízení pak neví, jestli informace dorazila na místo určení, nebo nikoliv. To má za následek určitou nespolehlivost přenosu informace, ale o mnohem méně režie potřebné pro přenos. V porovnání s TCP přenáší UDP datagram mnohém méně informací v záhlaví:



Obrázek 2.2: Uspořádání UDP datagramu
zdroj: `http://serverfault.com/a/8986`

Pro poslání informace potom stačí vyslat jeden tento datagram s daty a to je vše. Proto skutečnost, že neznáme výsledek přenosu vede k tomu, že musíme být smířeni s faktem, že se některé datagramy při přenosu jednoduše ztratí. Tento problém nastane až u větších sítí, ale při praktickém testování tohoto projektu se ukázalo, že může dojít ke ztrátě datagramů na levnějších komponentách sítě. V tomto případě mohl za ztráty switch. Tento nežádoucí stav může nastat buď vlivem malé paměti daného zařízení, nebo vysokou rychlostí posílání datagramů, což je hlavní příčina. Na levnějších zařízeních pak dochází k přenosu pouze nejrychlejšího náhodného datagramu. Celá síť se pak chová tak, že se datagramy posílají na náhodné prvky za tímto switchem, ale nikdy ne na více než jedno zařízení. Tento efekt byl pozorován na zařízeních ZyXEL ES-105A i TP-LINK TL-WR743ND. U switche D-Link DFE-916DX k tomuto efektu nedošlo.

Tento přenos (vzhledem k charakteru UDP) je tedy vhodný pro přenos velkého množství informací s tím, že nám případné ztráty nevadí. Typickým zástupcem toho typu přenosu jsou například kontinuální čidla, která neustále snímají (například teplotu) a v krátkých časových intervalech posílají informace na server.

3

Volba vhodné technologie

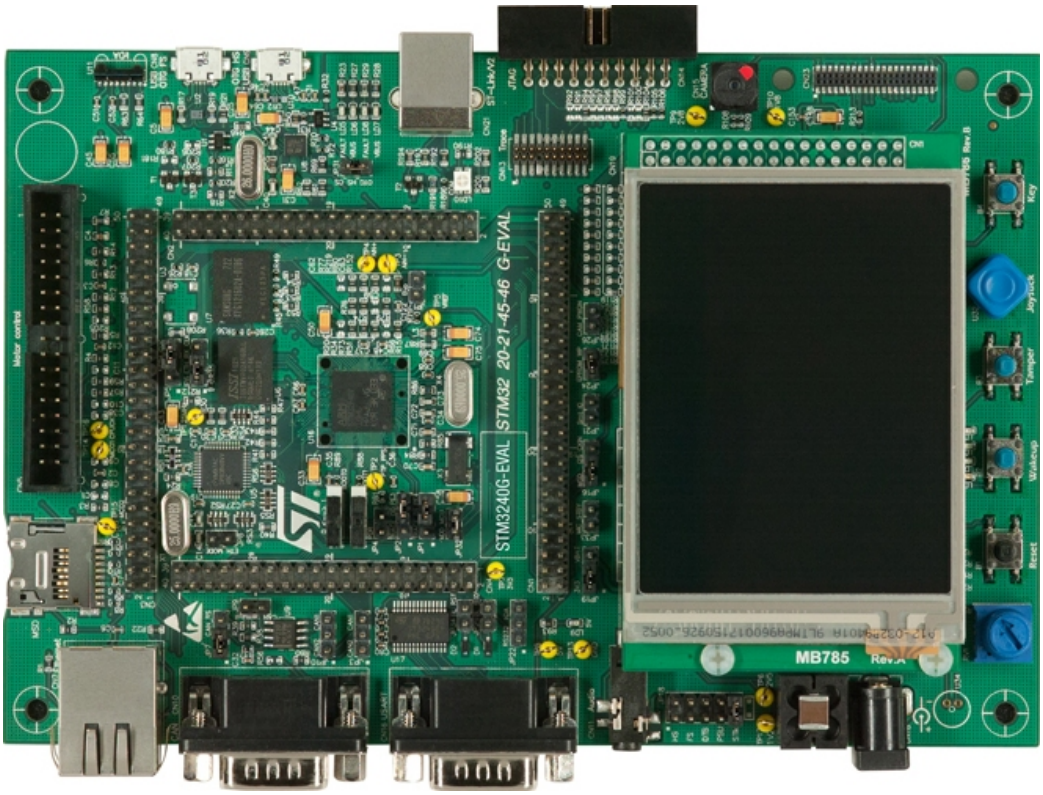
Pro tuto síť nejsou v tuto chvíli stanoveny zadávající firmou žádné konkrétní požadavky na hardware. Proto je možné vybrat z hlediska softwarového řešení jakoukoliv platformu. Z hardwarového hlediska je doporučeno používat vývojové desky od STMicroelectronics. V následující části budu popisovat jednotlivé použité technologie a důvod jejich volby.

3.1 Prvky senzorické sítě

Prvky senzorické sítě jsou testovány na vývojových deskách s mikrokontroléry STM32F207IGH6 a STM32F457IGH6 s využitím oficiálních Cube knihoven [4]. Jedná se o 32-bit mikrokontroléry pro obecné použití. Pro oba mikrokontroléry dále platí, že mají 176 pinů s velikostí flash paměti 1024 KB v provedení pouzdra UFBGA pro běžné rozsahy teplot od -40 do 85 °C.

3.1.1 Mikrokontrolér STM32F207IGH6

- Jádro ARM 32-bit Cortex™-M3 CPU (120 MHz max)
- 128 KB SRAM
- LCD interface
- 12 × 16-bit timer, 2 × 32-bit timer - až 120 MHz
- 15 komunikačních rozhraní
- 2 × USB, 10/100 Ethernet
- 8 až 14-bit interface pro kameru
- CRC jednotka



Obrázek 3.1: Použitá vývojová deska MB786 rev.B
zdroj: `http://bit.ly/1Kor8OH`

3.1.2 Mikrokontrolér STM32F457IGH6

- Jádro ARM 32-bit Cortex™-M4 CPU (168 MHz max)
- 192 KB SRAM
- LCD interface
- 12 × 16-bit timer, 2 × 32-bit timer - až 168 MHz
- 15 komunikačních rozhraní
- 2 × USB, 10/100 Ethernet
- 8 až 14-bit interface pro kameru
- CRC jednotka

Veškeré vývojové desky s mikrokontroléry jsou připojeny pomocí klasických síťových prvků a UTP kabelů do serveru.

3.2 Real-time server

Jako real-time server, který obsluhuje celou síť i webovou aplikaci, je zvolen Node.js [5]. Jedná se o platfomu postavenou nad V8 JavaScript Engine od společnosti Google. V8 je engine napsaný v C++, který využívá například prohlížeč Google Chrome jako jádro pro velmi rychlé zpracování JavaScriptu. Díky tomu je možné využívat téměř všech vlastností JavaScriptu, zejména pak single-thread asynchronní chování (neblokující I/O model) a EDA architektury. Jedná se o velmi podobný engine jako je v prohlížeči Google Chrome s tím rozdílem, že Node.js neobsahuje možnost práce s oknem programu nebo s dokumentem jako takovým, protože zde žádný není. Oproti tomu umožňuje přistupovat k `process` objektu, který zase není dostupný v prohlížečích. Následující ukázka ukazuje jednoduchý webový server napsaný právě s pomocí Node.js:

```
1  var http = require('http');
2  var PORT = 8000;
3
4  var server = http.createServer(function(request, response) {
5    response.writeHead(200, {'content-type': 'text/html'});
6    response.write("<h1>hello</h1>\n");
7    setTimeout(function() {
8      response.end("<p>world</p>\n");
9    }, 2000);
10 });
11
12 server.listen(PORT, function() {
13   console.log('Server is listening on port ' + PORT);
14 });
```

Tento server je zároveň vysoce škálovatelný, a to hlavně do šíře. Je tak možné vytvořit velký počet vzájemně komunikujících uzlů (node). Tyto uzly jsou však striktně odděleny (i na úrovni paměti) a nemohou se tak přímo ovlivňovat. Tento model je vhodný pro velmi vytížené aplikace, protože je možné potřebný výkon rozložit na velké množství méně výkonných strojů.

3.3 Databázový server

Na pozici databázového serveru byl zvolen Redis [6]. Redis je key-value databáze, ukládá si tedy hodnoty různých datových typů, ke kterým se

přistupuje pomocí identifikátoru - klíče. Od nejrozšířenějších relačních databázích se liší například tím, že je násobně výkonnější a neobsahuje relace. Významným prvkem této databáze je právě vztah klíče a hodnoty, kdy hodnotu je možné ukládat do sedmi datových struktur (string, hash, list, set, sorted set, bitmap, hyperloglog). Velkou rychlost Redisu zajišťuje zejména to, že pracuje s RAM pamětí serveru. Přijaté hodnoty si nejdříve ukládá právě do paměti a následně je podle konfigurace ukládá na disk. Příklad výchozí konfigurace:

```
# save <seconds> <changes>
save 900 1
save 300 10
save 60 10000
```

Vždy musí být splněna časová podmínka i množství změn za tento čas. Takové nastavení pak tedy udává, že se po 900 vteřinách (15 minut) uloží změny na disk, pokud byla provedena změna alespoň jednoho klíče, nebo po 300 vteřinách při změně alespoň deseti klíčů, resp. po minutě, došlo-li ke změně alespoň 10 000 klíčů. Již z této konfigurace je zřejmé, že se u Redisu očekává velké vytížení a tato databáze je na to připravena.

3.3.1 Redis klíče

Redis klíče mohou být velké až 512 MB a jsou binary safe, tzn. že platný je jak běžný text, tak prázdný text, ale také binární data nebo obsahy souborů (do maximální velikosti klíče). Samotným klíčům lze pak také nastavovat životnost (pomocí příkazu `EXPIRE`). Klíče potom vyprší po nastaveném čase.

3.3.2 Datová struktura string

Datová struktura string je nejzákladnější způsob jak je možné uložit data do databáze. Princip je velmi jednoduchý. Každému klíči se přiřadí textová hodnota. V této práci je datová struktura string používána k ukládání a inkrementování celkového počtu přijatých nebo odeslaných zpráv. Ukázka použití počítadla:

```
> SET counter 1
OK
> INCR counter
(integer) 2
> INCRBY counter 50
```

```
(integer) 52
> INCRBYFLOAT counter 3.0e3
"3052"
```

Časová náročnost těchto operací je O(1)[1].

3.3.3 Datová struktura hash

Hash je velmi podobný stringu s tím rozdílem, že je možné ukládat k jednomu klíči více hodnot. Časová složitost je pak O(1) pro jeden prvek, resp. O(N), kde N je počet ukládaných nebo vybíraných prvků. V této práci je datová struktura hash použita právě pro ukládání informací o koncovém zařízení. Konkrétně se o koncových zařízeních uchovává IP adresa, porty TCP a UDP komunikace, čas posledního ohlášení, status aktivity a počet přenesených zpráv.

3.3.4 Datová struktura list

List je v Redisu implementovaný jako spojový seznam. Tato implementace umožňuje vkládat nová data na začátek nebo na konec spojového seznamu v konstantním čase nezávisle na počtu prvků v seznamu. Časová složitost je tedy opět O(1). U operací, kdy se vkládá prvek dovnitř seznamu, je složitost O(N), kdy N je počet prvků, přes které je nutné iterovat, než se dostaneme k požadovanému umístění. V nejlepším případě muže být tedy i zde časová složitost O(1). List je v tomto projektu použitý pro ukládání historie příchozích dat z koncentrátorů následovně:

```
> LPUSH device_uid:data <data>
> LTRIM device_uid:data 0 999
```

Výhodné na tomto přístupu je to, že je složitost obou příkazů O(1). Je to dáno tím, že LPUSH umisťuje data do seznamu zleva, což není nijak časově náročné. Dále je složitost LTRIM funkce dána počtem prvků, které se touto funkcí odstraní, což je jeden starý prvek. S minimální režií je tak uchováváno posledních 1000 hodnot.

[1]Označení $O(f(x))$ značí asymptotickou složitost algoritmu. Složitosti algoritmů rozdělujeme do různých tříd $(1 < log(n) < n < n \cdot log(n) < n^k < k^n < k! < n^n)$, které určují, jak je daný algoritmus rychlý. Zápisy je pak možné číst tak, že algoritmus je stejně rychlý nebo rychlejší než $f(x)$.

3.3.5 Datová struktura set a sorted set

Sety jsou neseřazené kolekce unikátních stringů. Zde je nutné zdůraznit, že hodnoty zde skutečně nejsou seřazeny a vrácený výsledek tak může měnit pořadí. Výhodné je, že je možné do této množiny ukládat data se složitostí O(N), kde N je počet prvků. To samé platí i pro čtení. Tato struktura se hodí pro ukládání relací, čímž je možné simulovat relační vztahy mezi objekty. V tomto projektu je proto set používaný pro ukládání všech známých zařízení k síti a dále pro ukládání vazeb mezi jednotlivými koncentrátory.

Sorted set je obdoba setu s tím rozdílem, že prvky jsou v množině seřazeny podle zvoleného skóre. Tato struktura není v projektu nikde použita a je zde uvedena pouze pro úplnost.

3.3.6 Datová struktura bitmap

Bitmapy umožňují ukládat binární data do databáze. Tento způsob práce s daty je pouze obdobou práce se stringy. Vzhledem k tomu, že se jedná o práci s nejmenší jednotkou informace, je tato struktura velmi úsporná co se týče velikosti a velmi se hodí pro ukládání velkého množství pravdivých resp. nepravdivých informací. Bitmapy jsou limitovány na velikost 512 MB stejně jako klíče. To jinými slovy znamená, že je v jednom klíči možné uchovat informace až o 2^{32} stavech ($2^{32} = 4\,294\,967\,296\,b = 536\,870\,912\,B = 524\,288\,kB = 512\,MB$).

Tato struktura není v projektu nikde použita a je zde uvedena pouze pro úplnost.

3.3.7 Datová struktura hyperloglog

Posledním a relativně novým datovým typem je hyperloglog. Jedná se o statistickou datovou strukturu, která se používá zejména pro rychlé určení kvantity unikátních dat s chybou méně než 1%. Tato struktura předchází paměťové náročnosti při počítání množství unikátních dat. V běžném případě je totiž nutné pamatovat si tyto data, aby bylo možné při dalším vstupu unikátní hodnotu započítat, nebo ji ignorovat, protože je již započítána. Redis při své implementaci používá v nejhorším případě 12 kB paměti pro uchování této struktury.

Tato struktura není v projektu nikde použita a je zde uvedena pouze pro úplnost.

3.3.8 Výkon Redisu

Redis je díky svému přístupu k paměti a omezenému počtu zapisování na disk velmi rychlá databáze. Je připravena na škálování do šíře, takže je možné připojit další servery jako databázové uzly [7]. Níže je uvedený reálný výsledek benchmarku [8] na Linuxovém stroji Debian s 2×CPU po 2 GHz, 2 GB RAM. Test je proveden pro různé datové struktury a jejich příkazy vždy pro 50 souběžných připojení a 100 000 požadavků s délkou SET/GET hodnoty 256 B. V tomto prvním testu jsou vždy příkazy posílány postupně a postupně také vybavovány.

```
$ redis-benchmark -q -n 100000 -d 256
PING_INLINE: 212314.23 requests per second
PING_BULK: 211416.50 requests per second
SET: 131752.31 requests per second
GET: 199600.80 requests per second
INCR: 213219.61 requests per second
LPUSH: 213219.61 requests per second
LPOP: 204918.03 requests per second
SADD: 214592.28 requests per second
SPOP: 212765.95 requests per second
LPUSH (needed to benchmark LRANGE): 213675.22 requests per
 ↪ second
LRANGE_100 (first 100 elements): 45269.35 requests per second
LRANGE_300 (first 300 elements): 15586.04 requests per second
LRANGE_500 (first 450 elements): 9325.75 requests per second
LRANGE_600 (first 600 elements): 6472.49 requests per second
MSET (10 keys): 131578.95 requests per second
```

Je zřejmé, že již při základní konfiguraci dosahuje Redis vysokých výkonů. Nevýhodou této konfigurace, resp. přístupu k práci s Redisem, je skutečnost, že nový požadavek je vždy poslán až po zpracování databází a vrácení odpovědi. Redis totiž funguje pomocí TCP, takže klient odesílá požadavek a přijímá od serveru odpověď. Tato smyčka může být velmi krátká, zejména pak pokud je Redis umístěn na stejném serveru jako klient. V každém případě však vzniká časová prodleva mezi tím, kdy putují pakety od klienta k serveru a zpět. Tento čas se nazývá RTT. I čas potřebný pro lokální smyčku na serveru může být v součtu velký při velkém počtu požadavků. Tento problém se dá částečně vyřešit tzv. pipeliningem. Pak je možné posílat více zřetězených požadavků v jednom dotazu. Následující příklad ukazuje stejný benchmark jako dříve, ale se zapnutým pipeliningem. Vždy se zřetězí 16 příkazů do jednoho požadavku:

```
$ redis-benchmark -q -n 100000 -d 256 -P 16
PING_INLINE: 1612903.25 requests per second
PING_BULK: 2127659.75 requests per second
SET: 1086956.50 requests per second
GET: 1351351.38 requests per second
INCR: 1219512.12 requests per second
LPUSH: 934579.44 requests per second
LPOP: 1030927.81 requests per second
SADD: 1265822.75 requests per second
SPOP: 1562499.88 requests per second
LPUSH (needed to benchmark LRANGE): 990099.00 requests per
  ↪ second
LRANGE_100 (first 100 elements): 35186.49 requests per second
LRANGE_300 (first 300 elements): 8521.52 requests per second
LRANGE_500 (first 450 elements): 5236.70 requests per second
LRANGE_600 (first 600 elements): 3888.48 requests per second
MSET (10 keys): 207468.88 requests per second
```

3.3.9 RESP protokol

Redis databáze komunikuje interně přes TCP v RESP (Redis Serialization Protocol) formátu. RESP používá celkem 5 typů dat. Vždy platí, že první byte je byte určující o jaký formát se jedná:

- `+` simple string (jednoduchý řetězec)
- `-` error (chybový stav)
- `:` integer (celé číslo)
- `$` bulk string (binary safe řetězec)
- `*` array (pole)

Následuje samotný obsah nebo dodatečné informace, například o délce, a vše je ukončeno pomocí `CRLF` (`\r\n`). Postupně tedy přenášené informace mohou vypadat například takto:

- `+PONG\r\n`
- `-Error 123\r\n`
- `:54986\r\n`
- `$4\r\nPING\r\n` (první část určuje délku řetězce, NULL je pak `$-1\r\n`)

- `*2\r\n$3\r\nGET\r\n$3\r\nkey\r\n` (první je délka pole, následuje kombinace předchozích)

Redis server potom přijímá pole řetězců, které obsahují jednotlivé instrukce. Tento protokol je velmi důležitý, protože i koncentrátory (koncové členy sítě, se kterými server komunikuje) posílají data přes TCP i UDP v RESP formátu. Je tak možné data posílat přímo do databáze. Tato vlastnost však není využívána, protože je vhodné, aby byl jako prostředník server a například zjišťoval aktivitu koncentrátorů. Každopádně tato možnost zde je, a pokud by bylo zapotřebí ukládat data tou nejrychlejší cestou, přímý přístup do databáze je tímto možný a funkční.

3.4 Webová aplikace

Webový server je možné spustit na již běžícím Node.js serveru. Pro webovou aplikaci byl zvolen framework Sails.js [12]. Sails je MVC webový framework, který staví právě nad Node.js, ale ulehčuje práci při stavbě webových aplikací. Výhodou tohoto přístupu je to, že je možné na jednom serveru zapnout jak server zpracující požadavky z koncentrátorů, tak server obsluhující požadavky klientů z webového prohlížeče. Sails má navíc vestavěnou podporu pro protokol websocket, který je potřebný pro rychlou komunikaci serveru právě s prohlížečem.

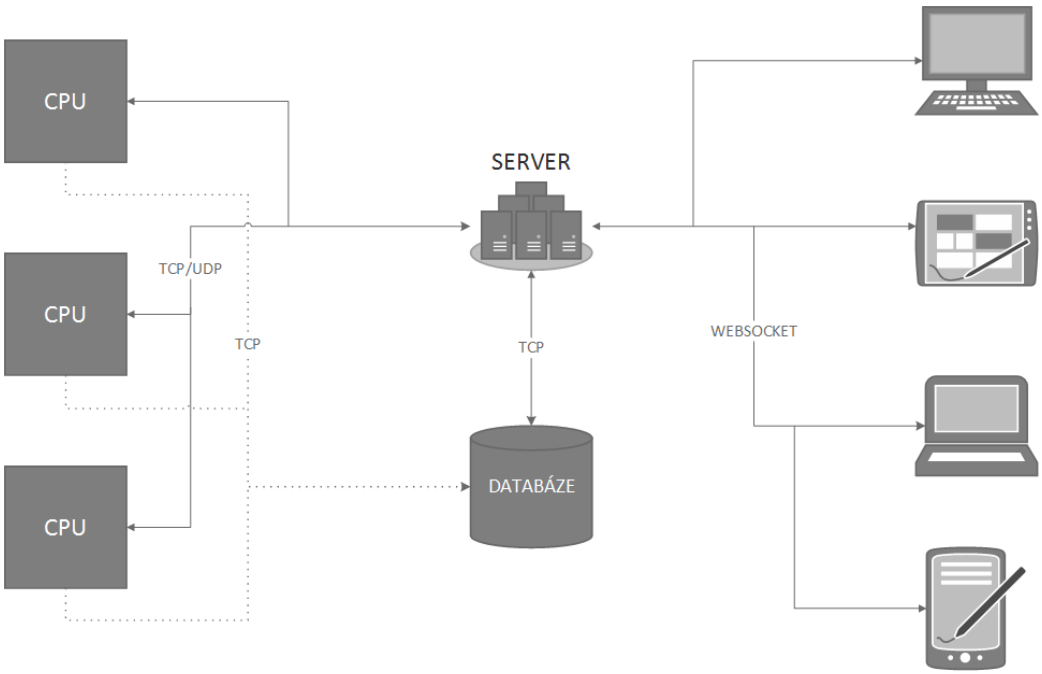
Volba tohoto frameworku je pouze na osobních preferencích. Žádný z existujících frameworků neobsahuje další výhody, které by jej stavěly do jednoznačné výhody. Navíc z hlediska sítě je potřebný hlavně Node.js a webová aplikace je možná postavit také pouze nad Node.js. Ačkoliv tedy tato aplikace tvoří velkou část kódů, není pro samotné fungování projektu klíčová.

4

Struktura programového řešení

Struktura celého programového řešení je znázorněna na obrázku 4.1. Celý projekt funguje následovně. Jednotlivé mikrokontroléry, resp. koncentrátory, snímají potřebné veličiny, nebo čekají na impulz od uživatele sítě popř. serveru. Mohou tedy aktivně odesílat snímané informace nebo reagovat na přijatý signál ze serveru. Tyto koncentrátory komunikují s real-time serverem pomocí protokolů TCP nebo UDP podle toho, jaký druh komunikace je pro daný účel potřeba. Server veškeré přijaté hodnoty ukládá do databáze a zároveň při každé příchozí akci prohlásí zařízení za aktivní. Udržuje tak neustále jeho status. Kromě toho, že server do databáze hodnoty ukládá, tak si také drží jednotlivé relace mezi koncentrátory, resp. koncovými členy, a odesílá data zpět. Tvoří tak jednotlivá propojení koncentrátorů, která je možné dynamicky měnit. Toto je asi největší výhoda tohoto řešení. Zároveň mají koncentrátory možnost zapisovat do databáze přímo. Je to dáno tím, že i se serverem komunikují v RESP formátu. Tato funkce není využívána, protože by bylo zapotřebí zapsat tvar databáze do každého koncentrátoru, což by bylo velmi omezující pro další rozšiřitelnost projektu. Nicméně tato možnost zde je a je možné ji využít pro rychlejší ukládání do databáze.

Na serveru dále běží webový server, který poskytuje webovou stránku, kde je možné celou síť ovládat. Každý uživatel sítě se pak může připojit pomocí svého zařízení (mobilní telefon, tablet, notebook, počítač, atd.) a síť ovládat. Je zřejmé, že je nutné umožnit pohodlné ovládání sítě, zároveň však musí být zamezena možnost změny konfigurace neautorizované osobě. Toto omezení je možné udělat na straně webové aplikace pomocí řízení uživatelských práv. Fakticky se jedná o změny relací v síti a o změnu parametrů jednotlivých koncových členů. Server se pak postará o distribuci dat v síti podle zvolené konfigurace.



Obrázek 4.1: Struktura programového řešení

4.1 Koncentrátory

Systémový firmware pro koncentrátory je napsán v programovacím jazyce C s využitím oficiálních Cube knihoven od STMicroelectronics. Jsou tady napsány nízkoúrovňově, ale se zachováním přijatelného programového prostředí. Koncentrátor má několik základních funkcí. Předně je jeho úkolem připojit se na pevně stanovenou IP adresu v síti. Ta je momentálně stanovena na `192.168.0.20:50000`. Koncentrátor se pak periodicky s frekvencí 1 Hz ohlašuje přes TCP serveru. Tato frekvence je zvolena libovolně s ohledem na rozumné vytížení sítě těmito jinak zbytečnými informačními pakety. Síť se tak zbytečně nevytěžuje a zároveň dochází k rychlému zaregistrování výpadku koncentrátoru. 1 Hz je navíc nejhorší případ, protože jakákoliv příchozí informace se zároveň považuje za ohlášení.

Struktura programu z pohledu adresářové struktury je následující. Zobrazeny jsou pouze důležité části programu:

```
concentrator
├── Drivers
│   ├── BSP ..... knihovny pro konkrétní hardware
│   ├── CMSIS ..... nízkoúrovňové definice
│   └── STM32F4xx_HAL_Driver ..... definice periferií
├── Inc ..... hlavičkové soubory
├── MDK-ARM ..... konfigurace projektu pro Keil
├── Middlewares ..... knihovny třetích stran (LwIP)
├── Src ..... zdrojové kódy aplikace
│   ├── app_ethernet.c ..... pomocný soubor pro práci s Ethernetem
│   ├── ethernetif.c ..... hlavní soubor pro práci s Ethernetem
│   ├── main.c ..... hlavní soubor aplikace
│   ├── stm32f4xx_hal_msp.c ..... konfigurace periferií
│   ├── smt32f4xx_it.c ..... obsluha přerušení
│   ├── tcp.c ..... soubor starající se o TCP komunikaci
│   ├── udp.c ..... soubor starající se o UDP komunikaci
│   └── ...
└── Utilities ..... doplňkové knihovny
```

Hlavní část této práce se nachází v `Src` složce, zejména se pak jedná o soubory `main.c`, což je soubor obsahující hlavní program. Tento program využívá funkce z dalších souborů, jmenovitě pak například `tcp.c` a `udp.c`, které se již podle názvu starají o TCP resp. UDP spojení se serverem. Následuje zjednodušený popis hlavního programu včetně několika ukázek kódu.

Nejdůležitější funkcí v každém podobném programu je funkce `main(void)`. Jedná se totiž o funkci, která je zavolána po spuštění programu. Ta prvně volá funkci pro inicializaci HAL knihovny `HAL_Init(void)` [9]. Následně je volána funkce pro nastavení systémových hodin `SystemClock_Config(void)`, konfigurace BSP (`BSP_Config(void)`), inicializace LwIP (`lwip_init(void)`), konfigurace síťového rozhraní (`Netif_Config(void)`), nastavení časovačů pomocí `TIM_Config(void)` a nastavení ADC převodníku (`ADC_Config(void)`). Toto nastavení zároveň všechny potřebné procesy startuje. Vždy, když dojde k nějaké akci, jako je například stisk tlačítka, zavolá se tzv. callback (například `HAL_GPIO_EXTI_Callback`), který je umístěn v souboru `main.c`. Zde se provedou další potřebné úkoly na základě této spouštěcí akce. Tím pádem není nutné mít veškerou logiku v hlavní smyčce programu. Tento princip je podrobněji rozebrán níže na příkladu diody a tlačítka.

Následující ukázka znázorňuje, jak Cube knihovny zabalují logiku jednotlivých operací. Pro jednoduchost uvádím práci s LED diodou. Samotná inicializace se provede velmi jednoduše pomocí `BSP_LED_Init(LED1)`. Z to-

hoto zápisu není zřejmé, co se fakticky děje. Nicméně jednou z výhod Cube knihovny je fakt, že používá velmi podobný princip jako je Dependency Injection, tedy předávání závislostí. Samozřejmě toto nemůže být dotaženo do dokonalosti jako u vyšších objektově zaměřených programů, ale podstatné je, že se identifikátor diody předává v parametru. Tato skutečnost zamezuje tomu, aby docházelo k magické inicializaci něčeho na pozadí. Samotná inicializační metoda pak vypadá následovně:

```
void BSP_LED_Init(Led_TypeDef Led) { //LED1 = 0
  GPIO_InitTypeDef  GPIO_InitStruct;

  LEDx_GPIO_CLK_ENABLE(Led); //LED1_GPIO_CLK_ENABLE

  GPIO_InitStruct.Pin = GPIO_PIN[Led]; //GPIO_PIN_6
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  GPIO_InitStruct.Speed = GPIO_SPEED_FAST;

  HAL_GPIO_Init(GPIO_PORT[Led], &GPIO_InitStruct); //GPIOG
}
```

Z toho plyne, že není nutné využívat tyto Cube funkce a je možné pracovat přímo s inicializačními strukturami. V tomto případě je to však zbytečné. U složitějších věcí, kde je zapotřebí upravit logiku inicializace, je naopak nevhodné tyto funkce používat. Pro úplnost, obsluha externího přerušení může vypadat například takto:

```
void EXTI15_10_IRQHandler(void) {
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_15); //Button
}
```

Ve funkci `HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)` je pak ukryta následující implementace:

```
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin) {
  if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != RESET) {
    //EXTI line interrupt detected
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
    HAL_GPIO_EXTI_Callback(GPIO_Pin);
  }
}
```

Nezbývá, než tedy implementovat funkci `HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)`:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  if(GPIO_Pin == GPIO_PIN_15) {
    BSP_LED_Toggle(LED3);
  }
}
```

Nyní se tedy po stisku tlačítka vyvolá přerušení, které zavolá patřičný callback. V tomto volání je možné implementovat jakoukoliv logiku, zde například jednoduché přepnutí stavu LED diody. Jedná se o jednoduché příklady, ale tento princip se dále velmi podobně opakuje dále. Případné implementační detaily lze pak dohledat například v referenčním manuálu [10].

4.1.1 Komunikace se serverem

Koncentrátory komunikují se serverem v RESP formátu, který může vypadat takto:

```
*2\r\n$4\r\nPING\r\n$11\r\nTEMP_000001\r\n
```

První část zprávy je samotný příkaz (PING) následovaný unikátním identifikátorem zařízení. Zpráva „PING“ nemá nic společného s datagramem „Echo Request“ protokolu ICMP [11], který se vyvolává ve většině OS právě pomocí příkazu `ping`. Název příkazu byl tímto protokolem pouze inspirován. Server tyto zprávy přijímá, parsuje a dále zpracovává. Konkrétně v tomto příkladu server přidá další zařízení, pokud neexistuje a již existující zařízení prohlásí za aktivní. Aktivita je vyhodnocována podle časové značky posledního ohlášení a prodlužuje se při každé zprávě, kterou koncentrátor odešle a server přijme. Konkrétně tento druh zpráv se posílá přes TCP s frekvencí 1 Hz. Následuje krátká vzorová ukázka funkčního kódu pro odesílání upozornění o aktivitě (`tcp.c`):

```
struct tcp_pcb *client_pcb;
struct ip_addr IPaddr;

client_pcb = tcp_new();
if (client_pcb != NULL) {
  IP4_ADDR( &IPaddr, IP_ADDR0, IP_ADDR1, IP_ADDR2, IP_ADDR3 );
```

```
7     tcp_bind(client_pcb, &IPaddr, DEST_PORT);
8     IP4_ADDR( &DestIPaddr, DEST_IP_ADDR0, DEST_IP_ADDR1,
       ↪ DEST_IP_ADDR2, DEST_IP_ADDR3 );
9     tcp_connect(client_pcb, &DestIPaddr, DEST_PORT,
       ↪ tcp_ping_callback);
10  } else { //can not create tcp pcb
11    memp_free(MEMP_TCP_PCB, echoclient_pcb);
12  }
```

Po připojení se zavolá funkce `tcp_ping_callback`, která již může odeslat data v požadovaném formátu. K tomu je možné využít soubor `resp.c`. Není to však nutné vzhledem k tomu, že formát zprávy zůstává stále stejný.

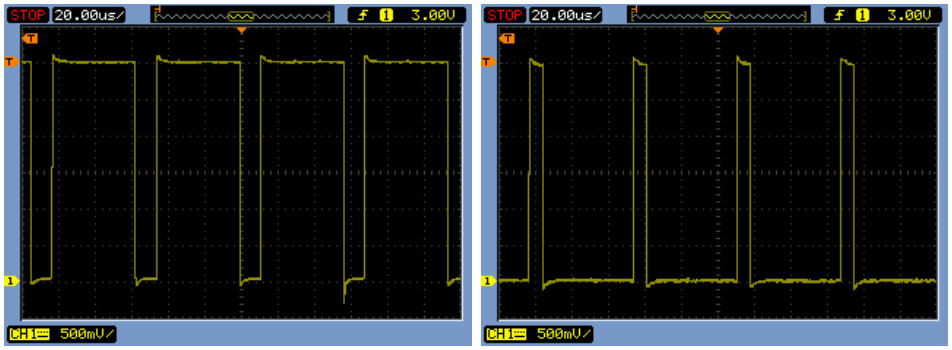
4.1.2 Příjem dat ze serveru

Pro příjem zprávy se serveru je možné použít funkci `tcp_echoclient_recv` pro TCP (`tcp.c`) nebo funkci `udp_receive_callback` pro UDP (`udp.c`), která je využívána častěji. V této funkci se berou data ze struktury, která je obsahuje. Dále je možné provádět podobné operace jako při přijetí na straně serveru, tedy rozparsovat zprávu a dále ji zpracovat. Zjednodušený příklad přijmutí dat a následného výpočtu procentuální hodnoty:

```
1  char *pc = (char *)p->payload; //struct pbuf *p
2
3  char *ptr;
4  uint16_t value = strtol(pc, &ptr, 10);
5
6  if(value <= 0) {
7    uhADCxConvertedValuePercent = 0;
8  } else if(value >= 1023) {
9    uhADCxConvertedValuePercent = 100;
10 } else {
11   uhADCxConvertedValuePercent = (value * 100) / 1023;
12 }
```

Proč se provádí přepočet na procenta? Celá síť si přeposílá číselné zprávičky v rozsahu 0 až 1023. Je to z toho důvodu, že pro každé zařízení má toto číslo jiný význam a koncentrátor se musí postarat o správnou interpretaci. V tomto případě má koncentrátor za úkol z přijaté hodnoty nakonfigurovat PWM výstup. Jak bude vysvětleno později, nejedná se o perfektní přístup, celá síť se však stává velmi jednotnou z hlediska přenášení informace a tyto

hodnoty je možné na serveru ovlivňovat. Pro jednoduchý příklad stmívače to například znamená, že je možné jedním kliknutím změnit funkci potenciometru z lineárního na exponenciální, logaritmickou nebo nějakou vlastní. To by bez zásahu serveru do této informace nebylo možné.



Obrázek 4.2: Změřené PWM výstupní signály

4.2 Real-time Server

Server je naprogramovaný v JavaScriptu s využitím Sails.js frameworku [12]. Tento framework staví nad Express frameworkem, který staví nad Node.js. Node.js je platforma postavená nad V8 [17], tedy JavaScriptovém jádře napsaném v C++. Toto jádro je například součástí prohlížeče Chrome. Kromě toho, že toto jádro dosahuje vysokého výkonu při zpracování JS, otevírá také zajímavé možnosti, jak přistupovat k programu. Konkrétně nabízí například event-driven chování nebo neblokující I/O model. Toto chování vychází z JavaScriptu jako takového. To může být svým způsobem zároveň nevýhoda. JavaScript se totiž chová (v běžných implementacích) jako jednovláknový asynchronní program. To sice umožňuje vytvářet zajímavé programové struktury, zároveň je však limitující jedno vlákno. Je proto lepší spustit odděleně webovou aplikaci a část zpracovávající příchozí signály. Každá tato část lze pak v případě potřeby spustit v clusteru [16]. Adresářová struktura je zobrazena níže. Opět jsou znázorněny jen důležité části. Tato struktura odpovídá běžné struktuře Sails.js [12] aplikace s EJS šablonovacím systémem.

```
/
├── redis ..... Redis server pro Windows
├── server ..... soubory real-time serveru
│   ├── .tmp ..... dočasné soubory aplikace
│   ├── api ..... modely a kontroléry aplikace
│   │   ├── controllers ..... kontroléry
│   │   ├── hooks ..... eventy
│   │   ├── models ..... modely
│   │   ├── policies ..... skripty upravující chování aplikace
│   │   ├── responses ..... HTTP odpovědi serveru
│   │   └── services ..... služby
│   ├── assets ..... obrázky, skripty a styly
│   ├── config ..... konfigurační soubory
│   ├── node_modules ..... nainstalované knihovny přes NPM
│   ├── tasks ..... úkoly pro Grunt
│   ├── views ..... šablony aplikace
│   │   ├── Device ..... šablony pro DeviceController.js
│   │   ├── Documentation ..... šablony pro dokumentaci
│   │   ├── Homepage ..... šablony pro HomepageController.js
│   │   ├── layout.ejs ..... layout pro šablony aplikace
│   │   └── ...
│   ├── app.js ..... script automatický start serveru
│   ├── Gruntfile.js ..... startovací skript pro Grunt
│   ├── package.json ..... JSON soubor pro NPM
│   └── ...
└── start.bat ..... startovací script
```

Celý server lze nastartovat souborem `start.bat`, jehož obsah je znázorněn níže.

```
1 @echo off
2
3 cd .\redis\bin\release\redis-2.8.17
4 start redis-server.exe
5
6 cd .\..\..\..\..\server
7 sails lift
8
9 exit
```

Je tedy zřejmé, že je nutné nastartovat Redis databázi a následně samotnou aplikaci pomocí příkazu `sails lift`. Při prvním spuštění je také nutné

doinstalovat závislosti pomocí NPM. Pokud by bylo nežádoucí spouštět aplikaci pomocí Sails.js, je možné použít klasický Node.js (`node app.js`). Samotný framework tedy neomezuje aplikaci a tu je tak možné přenést do jiného prostředí, jako je například Heroku [18], nebo využít podpůrné nástroje, jako je Forever [19] (`forever start app.js`), který zajistí, že program poběží nepřetržitě, tzn. i po fatální chybě.

Celá aplikace je rozdělena na několik částí. První jsou šablony jednotlivých stránek. Tyto šablony využívají šablonovací systém EJS, který má asi nejpřijatelnější zápis ze všech ostatních:

```
<h1><%= variable %></h1>
<ul>
  <% for(var i = 0; i < xarray.length; i++) { %>
    <li><%= xarray[i] %></li>
  <% } %>
</ul>
<%= img_tag('images/picture.png') %>
```

Do této šablony se předávají data z controlleru (zjednodušeně):

```
module.exports = {

  index: function (request, response) {
    RedisService.smembers('devices', function (err, result) {
      res.view({
        xarray: result,
        varibale: 'example'
      });
    });
  }

};
```

Při psaní takových programů je nutné mít na paměti, že se kód může (a bude) vykonávat asynchronně. Proto je nutné všechny synchronní operace provádět pomocí callbacků nebo například přes návrhový vzor promise. Rozdíly mezi těmito přístupy jsou popsány níže. Návrhový vzor promise ale není tolik rozšířený, takže se všude v tomto programu používá první přístup. Základní rozdíl je v tom, že běžné je v Node.js funkce kvůli synchronnímu chování zanořovat:

```
step1(function (value1) {
  step2(value1, function(value2) {
    step3(value2, function(value3) {
      step4(value3, function(value4) {
        console.log(value4);
      });
    });
  });
});
```

Zatímco s promise návrhovým vzorem zanořování není nutné:

```
Q.fcall(promisedStep1)
.then(promisedStep2)
.then(promisedStep3)
.then(promisedStep4)
.then(function (value4) {
  console.log(value4);
})
.catch(function (error) {
  console.error(value4);
})
.done();
```

Je možné použít knihovnu Q [20], ze které jsou tyto příklady přebrány. Toto chování by mělo být v pozdějších verzích JavaScriptu k dispozici bez potřeby knihoven třetích stran. Vzhledem k tomu, že použití callbacků má celou řadu nevýhod, návrhový vzor promise se stává velmi populárním a postupně na něj přecházejí všechny velké knihovny.

Samotná aplikace využívá následující tabulky v Redis databázi, kde `xxx` je unikátní identifikátor zařízení:

- `devices` (set) - obsahuje unikátní identifikátory připojených zařízení
- `device:xxx` (hash) - informace o konkrétním zařízení:
  - `ip` - IP adresa zařízení
  - `port` - TCP port
  - `active` - logická hodnota určující, jestli je zařízení aktivní
  - `last_ping` - čas posledního ohlášení
  - `msg_count` - počet vyměněných zpráv se serverem

  - `udp_port` - UDP port
  - `function` - zvolená funkce pro přepočet hodnot

- `xxx:data` (list) - historie přijatých dat ze zařízení (posledních 1000 hodnot)
- `xxx:table` (hash) - vypočtená převodní tabulka
- `connection:xxx` (set) - tabulka uchovávající relace

Dále je pak k dispozici hodnota `msg_count`, která udržuje počet vyměněných zpráv s jakýmkoliv zařízením připojeným do sítě.

4.2.1 Komunikace s koncentrátory

Server komunikuje s koncentrátory hlavně pomocí UDP. Celá logika je umístěna v souboru `api/hooks/UDP/index.js`, kde je také vyřešené odeslání UDP datagramů. Tento script je ve formě tzv. hooku. Volně by se tento termín dal přeložit jako „hák“, což jej přesně vystihuje. Tento kód se totiž navěsí na okamžik spouštění celé aplikace ještě před nastartováním webového serveru. Je zde použit callback přístup, nikoliv návrhový vzor promise. Z důvodu složitosti zde neuvádím žádné ukázky kódu, za zmínku však stojí jedna zajímavá vlastnost, která je společná pro server i koncentrátory. Ve skutečnosti totiž neodesílají data pořád, ale pouze, pokud je to potřeba, tzn. pokud se nová informace liší od předchozí. Tímto se zásadně sníží počet přenášených datagramů a je možné zvýšit rychlost cyklu. Je potřeba myslet na to, že nikdy nenastane dlouhá doba, kdy by se koncentrátor úplně odmlčel. Koncentrátor se totiž pravidelně ohlašuje serveru, čímž říká, že je stále aktivní a správně funguje. Právě na tomto místě pak dochází k využívání převodních tabulek, které jsou podrobně popsány v další kapitole. Podobná logika je k nalezení i v souboru `api/hooks/TCP/index.js` pro TCP. V souboru `api/hooks/routine.js` dochází k vyhodnocování aktivity připojených zařízení na základě posledního ohlášení.

4.2.2 Webový server

Kromě real-time serveru existuje také webový server, který zabírá ve složce `server` nejvíce místa. Jednou z hlavních složek tohoto serveru, kromě již zmíněných šablon a kontrolérů, je část obsluhující websocket. Ta je k nalezení v souboru `api/hooks/websocket.js`. V tomto projektu je použita část frameworku Sails.js [12], která websockety obsluhuje, fakticky však pouze zabaluje knihovnu Socket.IO [21]. Následuje zjednodušená ukázka použití této

knihovny a to včetně zabalení kódu tak, aby se choval jako hook a spustil se při startu aplikace:

```
module.exports = function WebsocketHook(sails) {
  return {
    start: function () {
      sails.io.on('connection', function (socket) {
        RedisService.smembers('devices', function (err, result) {
          socket.emit('devices', result);
        });
      });
      sails.log('Starting WEBSOCKET server...');
    },
    initialize: function (cb) {
      var hook = this;
      hook.start();
      return cb();
    }
  }
};
```

Sails.js při startování serveru zavolá metodu `initialize`. Ta následně spustí požadovaný kód, v tomto případě odesílá při připojení klienta informace o všech zařízeních, které jsou k dispozici. Právě v této části serveru dochází k další zajímavé vlastnosti celého systému. Momentálně se totiž všechny informace v síti posílají jako desítková celá čísla v rozsahu 0-1023. Tzn., že například výstupem z ADC je jedna z hodnot v tomto rozsahu. Výhodné je to, že je možné jednoduše tuto informaci na serveru upravovat. Jednou z úprav jsou například nelinearity. Další je však jednoduchý klouzavý průměr (SMA). Protože příchozí data se nepatrně mění v jednotkovém rozsahu například vlivem nečistot potenciometru, je nutné tuto informaci vyhladit. Toto vyhlazení je pouze z estetických důvodů, protože vzhledem k tomu, jak je celý systém rychlý, toto přeskakování se dostane i do uživatelského rozhraní a to nepůsobí dobře. Konkrétně se vypočítává SMA z posledních pěti hodnot:

$$SMA = \frac{X_t + X_{t-1} + X_{t-2} + X_{t-3} + X_{t-4}}{5} \tag{4.1}$$

Klouzavý průměr se tedy používá pouze při výstupu do webové aplikace. Jinak funguje celá síť s takovou informací, jaká opustila koncentrátor. Je nutné myslet na to, že klouzavý průměr sice vyhlazuje výkyvy v průběhu,

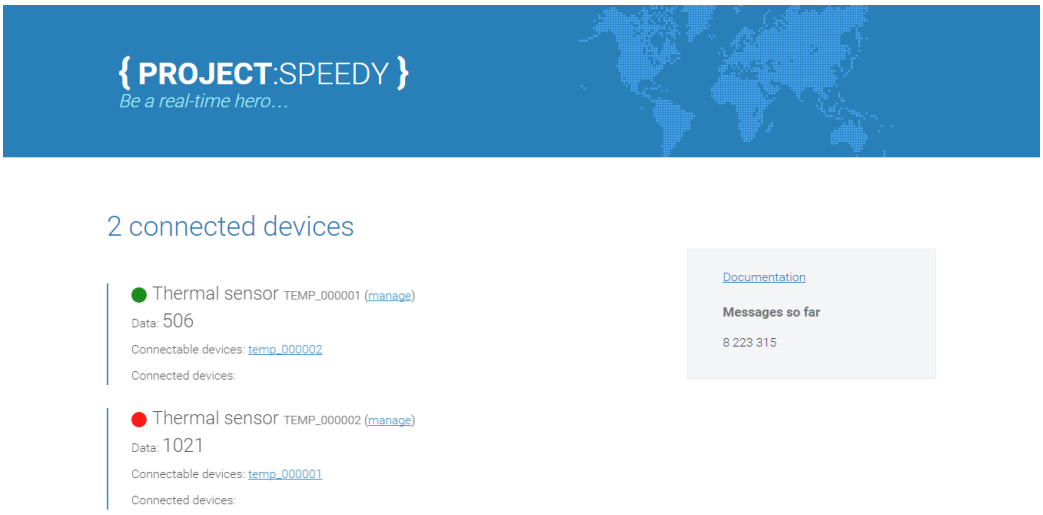
zároveň však celý průběh zpomaluje. Toto zpomalení je pak větší se zvětšujícím se počtem vzorků, které se do SMA zahrnují.

Velkou výhodou je fakt, že je celý systém možné ovládat z webové aplikace. Tu je možné otevřít téměř na jakémkoliv zařízení, které je připojené k internetu. Celou aplikaci je však ještě možné zabalit do nativní mobilní aplikace například pomocí Apache Cordova [22], takže ji lze na jakékoliv mobilní zařízení nainstalovat. Nejedná se sice o nejlepší způsob jak takovou aplikaci postavit, je to však mnohem rychlejší a levnější cesta. Některé frameworky, jako například Meteor umožňují tyto aplikaci vystavit již v základu pomocí dvou příkazů [23]. Tento framework však nebyl zvolen, protože nemá dobrou integraci Redis [6] databáze a hodí se na jiné aplikace, kde je možné využít principu „latency compensation“, tedy kompenzace času spotřebovaného pro komunikaci mezi serverem a prohlížečem. Toho je docíleno tak, že se programové metody umístí jak na server, tak na klientskou stranu. Následně při požadavku se metody zavolají na serveru, ale framework na straně klienta nečeká a provede simulaci dané operace i v prohlížeči. Tak je možné některé operace zpracovat ještě před odpovědí serveru. Pokud se výsledek operace ze serveru shoduje s výsledkem na straně klienta, vše je v pořádku. V opačném případě se provedené změny vrátí zpět a prohlížeč skutečně zrcadlí stav serveru. V případě úspěšné operace se tedy aplikace chová velmi rychle, jinak může docházet ke zvláštnímu chování uživatelského prostředí. Takto samozřejmě není možné provádět operace čekající na výsledek z databáze.

5

Praktická aplikace

Vzhled a ovládání celé aplikace je velmi jednoduché a intuitivní. Na úvodní stránce je přehled všech zařízení zapojených do sítě včetně jejich stavu. Vedle názvu zařízení je vidět, jestli je online, či nikoliv. Pod názvem je vidět aktuální přijatá informace v číselné podobě. Následuje seznam připojených a připojitelných zařízení.



Obrázek 5.1: Úvodní stránka aplikace - přehled připojených zařízení

Jedním kliknutím je možné propojit jakékoliv koncentrátory. V tu chvíli začne server přeposílat příchozí data na všechna připojená zařízení. Daným zařízením posílá pouze ty informace, které jim posílat má. To je dáno právě

tím, jak jsou zařízení vzájemně propojena. Počet připojitelných zařízení není nijak omezen. Toto je velká výhoda celého projektu. Není totiž vázán na fyzická spojení a příchozí data je tak možné poslat všem v síti bez dalších technických komplikací. Ačkoliv Redis [6] není relační databáze, tak se v ní mimo jiné uchovávají hlavně relace. Relační databáze nebyla zvolena hlavně proto, že nedosahují takových výkonů, jako právě Redis.

Detail konkrétního zařízení poskytuje podobný pohled, navíc však ukazuje dodatečné informace jako jsou například IP adresy nebo čas posledního ohlášení. Kromě samotné vizualizace historie příchozích dat je možné nastavit charakter dat odchozích. To znamená, že když bude připojeno další zařízení, nebudou se data rovnou přeposílat, ale najde se příslušná hodnota podle zvolené funkce a tato hodnota se pošle. Ve výchozím stavu se vše chová lineárně, tzn. jaká informace přijde, taková se přeposílá. Je však možné zvolit exponenciální (rovnice 5.1), logaritmický (5.2) nebo vlnitý charakter (5.3).

$$y_{\_exp} = 1.80753 \cdot 1.00625^{x} \tag{5.1}$$

$$y_{\_log} = -1053.96 + 289.931 \cdot ln(x) \tag{5.2}$$

$$y_{\_vlna} = -3.23206 \cdot 10^{-8} \cdot x^4 + 0.000068 \cdot x^3 - 0.044362 \cdot x^2 + \\ + 9.59513 \cdot x - 47.9076 \tag{5.3}$$

$$y_{\_bool} = \begin{cases} 0 & \text{pro } x \leq 512 \\ 1023 & \text{pro } x > 512 \end{cases} \tag{5.4}$$

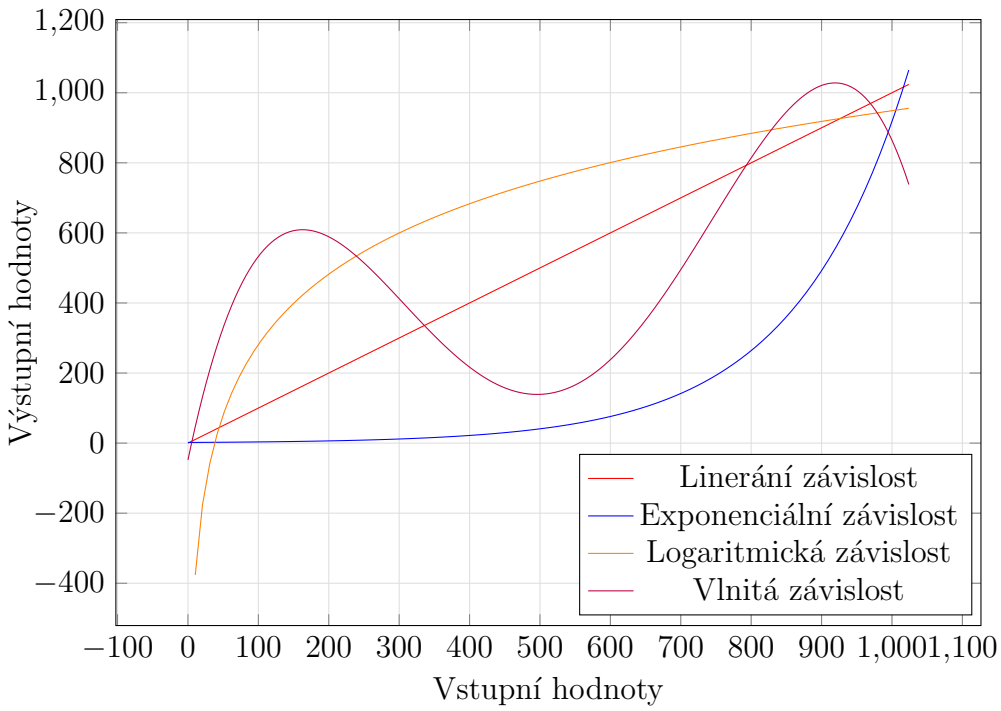
Tyto funkce jsou výsledkem aproximace ručně zvolených funkcí. Byl kladen důraz na to, aby funkce v celém svém rozsahu měly nějakou rozumnou hodnotu. Zároveň jsou v projektu zařazeny i zdánlivě nesmyslné funkce (např. rovnice 5.3), ty jsou zde pro ukázání možností systému. Poslední variantou je boolean závislost, kdy do hodnoty 512 včetně je výstupem 0, jinak maximální hodnota. Lze ji tedy definovat jako upravenou Heavisideovu funkci viz rovnice 5.4.

Tyto funkce jsou naprogramovány ve složce `api/services` v souboru `FunctionsService.js`. Ukázková implementace logaritmické funkce vypadá takto:

```
1 logarithmic: function (device, after_callback) {
2   RedisService.del(device + ':table');
3   for (var iterator = 0; iterator <= 1023; iterator++) {
```

```
4       var entry = -1053.96 + (289.931 * (Math.log(iterator) /
          ↪ Math.log(Math.exp(1))));
5       entry = entry <= 0 ? 0 : entry;
6       entry = entry >= 1023 ? 1023 : entry;
7       RedisService.hmset(device + ':table', iterator,
          ↪ Math.round(entry));
8     }
9     after_callback();
10  },
```

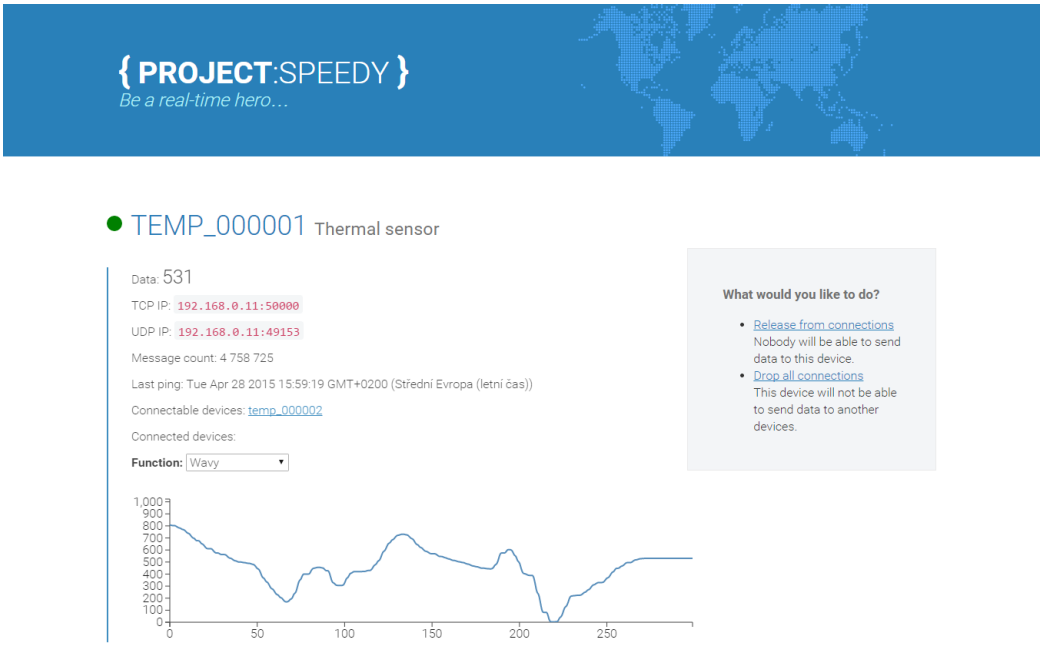
Funkce tedy nejdříve maže předchozí převodní tabulku a následně ukládá nové vypočtené hodnoty. Pomocí ternárního operátoru [25] jsou pak implementovány omezovače hodnot.



Obrázek 5.2: Převodní funkce vstupních hodnot

Cílem této ukázky je ještě jednou znázornit, že celý systém pracuje na základě přeposílání jasně daných informací, které jsou realizovány pomocí jednoduchých čísel. Je tak možné provádět jakékoliv matematické operace bez nutnosti znalosti významu této informace. Tento přístup však není ideální. V první řadě se až postupem času ukázalo, že rozsah 1024 hodnot je

pro určité případy malý. To se projevuje u rychlých výstupů (například u diody). Jde totiž o to, že v určité hodnotě má výstup, na který je připojená dioda, jedno efektivní napětí a o jeden krok dále je toto napětí o mnoho větší, protože takový je vypočtený výstup z převodní funkce (hodnota 300 $\Rightarrow 1,3\,V$ ale 301 $\Rightarrow 1,5\,V$). I při nejpomalejších změnách hodnot jsou tyto kroky rozpoznatelné, rychlost zde nehraje hlavní roli. Jedná se o nepatrné změny, jsou však postřehnutelné. A to tím více, čím je nárůst hodnot strmější, tedy jak moc velká chyba vzniká v převodní tabulce. Jedno z možných řešení je zjemnění celého rozsahu nebo neumožnění velkých skokových změn, například pomocí některého z druhů klouzavého průměru. Je však zapotřebí myslet na to, že i tyto úpravy mají své nevýhody. V prvním případě množství hodnot a stále větší kmitání kolem jedné hodnoty, v druhém případě zpomalení systému.



Obrázek 5.3: Detailní pohled na připojené zařízení

Jedním z možných vylepšení je tak dvojité chování serveru. Ten by totiž mohl přijímat jak převedené hodnoty, tak obecné hodnoty z čidla. Server by jim sice nemohl rozumět (pokud by neměl sám implementovanou převodní tabulku), ale mohl by informaci přeposílat dále do sítě. Tato funkce v současné době není implementovaná a to hlavně z toho důvodu, že je náročné tyto

informace nějakým způsobem využívat a skladovat v databázi, protože se jedná o velmi konkrétní problémy vázané na konkrétního výrobce zařízení nebo čidla. Oproti tomu číselná interpretace hodnot dobře ukazuje, co je v této síti možné vytvořit.

Vzhledem ke svým vlastnostem je možné tuto síť použít jako domovní síť pro ovládání běžných prvků, což byl prvotní účel. Z hlediska technologie však tato síť není limitována pouze na ovládání elektroinstalace, ale je možné ji využít pro jakoukoliv senzorickou síť pro sběr dat a zároveň ovládání koncových členů. Její rozumné využití by bylo však tam, kde se bude struktura sítě často měnit nebo zařízení přesouvat.

6

Rozšíření stávajícího řešení

Stávající řešení je plně funkční a splňuje veškeré požadavky v zadání. Jedná se však pouze o základ, na kterém lze stavět systém, který by bylo možné použít v reálných budovách. Prvně je totiž zapotřebí tuto síť zabezpečit. To se týká zejména okamžiku, kdy by síť začala komunikovat přes Wi-Fi (nebo jinou bezdrátovou technologii), ale platí to stejně i pro metalické vedení. Nesmí být možné, aby mohl kdokoliv ovlivňovat chování sítě, pokud k tomu není oprávněn.

Jak bylo již řečeno, samotná myšlenka není nijak limitována na přenos informace pomocí vodičů a je možné použít bezdrátovou komunikaci. Jednou ze zajímavých způsobů bezdrátové komunikace, ačkoliv také možná poněkud futuristickým, je Li-Fi [24]. Tuto komunikaci poprvé představil prof. Harald Haas v roce 2011 v Edinburghu při vystoupení na konferenci TEDGlobal [26]. Jedná se o přenos informace pomocí viditelného světla. Tento přístup má celou řadu výhod. Kromě kapacity a efektivnosti stojí za zmínku hlavně fakt, že každý v budovách svítí a je tedy pro tento přenos informací vlastně připraven. V neposlední řadě se jedná o bezpečný přenos, a to jak z hlediska lidského zdraví, tak i z hlediska různých nežádoucích odposlechů, protože se informace šíří pouze tam, kam dané světlo svítí (nikoliv např. skrz zeď). Síla a dosah signálu jsou tedy doslova vidět.

Dalším důležitým prvkem je implementace IPv6. Tyto adresy je možné využívat již od verze 1.4.x LwIP odděleně od IPv4. V pozdějších verzích by mělo být možné použití IPv4 a IPV6 současně. V současné chvíli je totiž nepsaným předpokladem, že budou koncentrátory připojeny v privátní síti a využívají IPv4. Pokud by však měla síť fungovat i na veřejné síti, vzroste počet potřebných IPv4 adres a již v tuto chvíli je jich nedostatek. Oproti tomu je IPv6 adres 2^{128} [13][1], což je více než dostatek. V tomto projektu

[1] Ve skutečnosti je jich o něco méně viz článek od Chris Welsh „Just how many IPv6 addresses are there? Really?“ (`http://bit.ly/1Js6EpZ`)

je použit LwIP stack, který IPv6 podporuje. Tato vlastnost není implementována, protože není potřeba. Pokud by se však projekt rozrostl do větších rozměrů, bylo by jej vhodné směřovat do stavu tzv. „fog computingu“. To znamená, že se z původně převážně centralizovaného systému začne stávat silně distribuovaný a původně centralizovaná část sítě bude sloužit pouze pro analýzy a statistiky. Veškeré zpracování dat se bude odehrávat na krajích sítě. Tím se vyřeší například problém s latencí. Zde by již bylo krátkozraké uvažovat překlad IP adres v rámci intranetové sítě, protože jednotlivými koncovými členy sítě mohou být jakákoliv připojitelná zařízení, tedy například automobily, mobilní senzory atd. Vzhledem k tomu, že je v současné chvíli celá síť závislá na centrálním serveru, nelze tento požadavek jednoduše implementovat. Bylo by však vhodné, aby se server začal postupně přesouvat na samotné koncentrátory, až by jej vůbec nebylo potřeba. To by znamenalo server úplně horizontálně rozškálovat, což v současnou chvíli není možné. Jednak proto, že by se to z hlediska Node.js nedělalo dobře, jednak také proto, že koncentrátory mají poměrně malý výkon. Malý výkon v tom smyslu, že pro rozumné spuštění Node.js nebo konkurenčního io.js je nutné Linuxové prostředí. Nicméně reálně fungující projekt využívající OpenWrt Linux [14] s io.js je například Tessel 2 (Cortex™-M3 CPU - 180 MHz) [15]. Tento krok by přiblížil celý projekt k naprosto autonomní síti, kde by se velmi jednoduše řešil například výpadek jednoho z koncentrátorů. Přestala by totiž fungovat pouze malá část sítě. Navíc by bylo možné částečně se zbavit metalických vodičů a vytvářet tzv. mesh sítě, což by ostatně bylo žádoucí. Každý koncentrátor by se mohl bez větší námahy připojit na všechny koncentrátory, které jsou poblíž.

Dále je zajímavou myšlenkou implementovat real-time přenos i na komunikaci mezi koncentrátory a serverem (např. pomocí Ethernet Powerlink). Tato vlastnost nebyla implementována ze dvou důvodů. Jednak to nebylo vzhledem k zadání žádoucí a dále při konzultaci se zadávající firmou byl stanoven závěr, že by tato náročná implementace neměla tak velký dopad, aby stálo za to real-time komunikaci v tomto slova smyslu řešit. V závěru práce se však ukazuje, že by jakákoliv real-time komunikace byla přínosem. Celý systém totiž sice funguje, ale neexistují žádné jasně dané časové vztahy, takže se může stát, že se informace někde zdrží. To není uživatelsky přívětivé. Je však nutné zdůraznit, že tento efekt byl způsoben hlavně tím, že byla celá práce vyvíjena na běžném notebooku s OS Windows a to pro tuto aplikaci není nejvhodnější. Vhodné prostředí je jednoznačně Linuxové. Docházelo pak k tomu, že server (notebook) nezvládal vybavovat všechny požadavky plynule. V současné chvíli také není kvůli jednoduchosti implementováno ani přijímání adres z DHCP serveru. Na funkcionalitě se nic nemění, je však možné pohodlně vyvíjet i bez nutnosti dalšího prvku v síti.

Velmi aktuální novinkou je potom představení nového operačního systému Brillo od společnosti Google na konferenci Google I/O 2015 , která proběhla 28. až 29. května v San Franciscu. Úkolem tohoto systému by mělo být sjednocení komunikace pro IoT zařízení pomocí nového protokolu Weave. Brillo by mělo být odvozeno od OS Android s tím, že bude přizpůsobený pro málo výkonný hardware. Svojí koncepcí je velmi podobný architektuře Cube knihoven. Je však zřejmé, že nepůjde ani tak o jádro a HAL vrstvu, ale spíše o samotnou komunikaci pomocí Weave (formát JSON). Zejména pak z toho důvodu, že tomuto protokolu bude zřejmě OS Android jako takový už v základu rozumět. Tento projekt by měl být uveřejněn na konci roku 2015. Stálo by proto za zvážení implementace protokolu Weave. Z hlediska JavaScriptu se jedná o přirozený formát, u koncentrátorů tomu tak není. Zatím však nejsou dostupné žádné podrobnější informace.

V neposlední řadě bude také nutné vybavit síť velkým počtem různorodých prvků jako jsou různé vypínače, snímače a akční členy, protože dobrou síť dělá mimo jiného také počet možností, které lze se sítí dělat.

7

Závěr

Tato práce byla pro mě velkým přínosem zejména díky tomu, že jsem si mohl prakticky vyzkoušet JavaScriptový real-time server Node.js v kombinaci s key-value databází Redis. Vzhledem ke svým speciálním vlastnostem není moc aplikací, kde lze tyto technologie použít. Celý efekt je umocněn tím, že jsem mohl pracovat s mikrokontroléry od STMicroelectronics. Spojení webové aplikace a elektroniky je v dnešní době stále velmi nezvyklé, ale pomalu nabírá na popularitě.

V práci se podařilo úspěšně vyřešit veškeré body zadání. Vedle teoretického rozboru a výběru vhodných technologií byla celá myšlenka naprogramována a několik stovek hodin postupně testována. Celkem bylo mezi koncentrátory a serverem vyměněno více než 32 220 000 paketů nesoucích data, což odpovídá zhruba $1,61\,GB$ přenesených testovacích dat při průměrné velikosti paketu $50\,B$. Při tomto testování se potvrdila původní myšlenka, tedy že ovládat síť pomocí přenášené informace má mnohem více možností než ovládání toku energie v rozvodu a je možné něco takového prakticky zrealizovat.

Hlavním nedostatkem a překážkou na cestě k širšímu praktickému využití je nutnost vytvoření zcela nových ovládacích prvků v běžných elektrorozvodech. To se týká prakticky jakéhokoliv zařízení, protože tato myšlenka počítá s tím, že bude možné s libovolným prvkem sítě komunikovat a předávat si informace. Toto je však překážka, která bude vzhledem ke vzrůstající popularitě IoT brzy překonána. Tato síť byla v práci navržena a vyzkoušena jako metalická. Samotný návrh však není na vodiče nijak vázaný a lze jej využít v kombinaci s bezdrátovou komunikací například na principu Wi-Fi nebo Li-Fi.

Literatura

[1] FETTE, Ian a Alexey MELNIKOV. The WebSocket Protocol. *The Internet Engineering Task Force* [online]. 2011 [cit. 2015-05-20]. Dostupné z: `https://tools.ietf.org/html/rfc6455`

[2] LAMMERMANN, Sebastian. Ethernet as a Real-Time Technology. *Lammermann.eu* [online]. 2008 [cit. 2015-05-20]. Dostupné z: `http://www.lammermann.eu/wb/media/documents/real-time_ethernet.pdf`

[3] SOSINSKY, Barrie A. *Mistrovství – počítačové sítě*. Vyd. 1. Brno: Computer Press, 2011, 840 s. Mistrovství (Computer Press). ISBN 978-80-251-3363-7.

[4] *Getting started with STM32CubeF2 firmware package for STM32F2xx series: User manual* [online]. 2014 [cit. 2015-05-20]. Dostupné z: `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/DM00111485.pdf`

[5] *Node.js* [online]. [cit. 2015-05-20]. Dostupné z: `http://nodejs.org/`

[6] *Redis.io* [online]. [cit. 2015-05-20]. Dostupné z: `http://redis.io/`

[7] Redis cluster tutorial. *Redis.io* [online]. 2015 [cit. 2015-05-20]. Dostupné z: `http://redis.io/topics/cluster-tutorial`

[8] How fast is Redis?. *Redis.io* [online]. [cit. 2015-05-20]. Dostupné z: `http://redis.io/topics/benchmarks`

[9] *Description of STM32F4xx HAL drivers: User manual* [online]. 2015 [cit. 2015-05-20]. Dostupné z: `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/DM00105879.pdf`

[10] *Reference manual - STM32F405xx/07xx, STM32F415xx/17xx, STM32F42xxx and STM32F43xxx advanced ARM®-based 32-bit MCUs* [online]. 2015 [cit. 2015-05-20]. Dostupné z: `http:`

//www.st.com/web/en/resource/technical/document/reference_manual/DM00031020.pdf

[11] POSTEL, Jon. Internet Control Message Protocol. *The Internet Engineering Task Force* [online]. 1981 [cit. 2015-05-27]. Dostupné z: http://tools.ietf.org/html/rfc792

[12] *Sails.js* [online]. [cit. 2015-05-20]. Dostupné z: http://sailsjs.org/

[13] Understanding IP Addressing. *RIPE NCC* [online]. [cit. 2015-05-20]. Dostupné z: http://bit.ly/1JvOvaO

[14] Linux distribution for embedded devices. *OpenWrt* [online]. [cit. 2015-05-20]. Dostupné z: https://openwrt.org/

[15] *Tessel 2* [online]. [cit. 2015-05-20]. Dostupné z: https://tessel.io/

[16] Node.js v0.12.3 Manual & Documentation. *Cluster* [online]. [cit. 2015-05-20]. Dostupné z: https://nodejs.org/api/cluster.html

[17] *V8 JavaScript Engine* [online]. [cit. 2015-05-20]. Dostupné z: https://code.google.com/p/v8/

[18] *Heroku* [online]. [cit. 2015-05-20]. Dostupné z: https://www.heroku.com/

[19] A simple CLI tool for ensuring that a given script runs continuously. *Forever* [online]. [cit. 2015-05-20]. Dostupné z: https://github.com/foreverjs/forever

[20] A tool for creating and composing asynchronous promises in JavaScript. *Q* [online]. [cit. 2015-05-20]. Dostupné z: http://documentup.com/kriskowal/q/

[21] *Socket.IO* [online]. [cit. 2015-05-20]. Dostupné z: http://socket.io/

[22] Platform for building native mobile applications using HTML, CSS and JavaScript. *Apache Cordova* [online]. [cit. 2015-05-20]. Dostupné z: http://cordova.apache.org/

[23] Running your app on Android or iOS. *Meteor* [online]. [cit. 2015-05-20]. Dostupné z: https://www.meteor.com/try/7

[24] *pureLiFi* [online]. [cit. 2015-05-20]. Dostupné z: http://purelifi.com/

[25] VRÁNA, Jakub. *1001 tipů a triků pro PHP*. Vyd. 1. Brno: Computer Press, 2010, 136 Co je ternární operátor. ISBN 978-80-251-2940-1.

[26] prof. HAAS, Harald. Wireless data from every light bulb. *TED - Ideas worth spreading* [online]. 2011 [cit. 2015-05-20]. Dostupné z: `http://www.ted.com/talks/harald_haas_wireless_data_from_every_light_bulb`

Rejstřík