VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

SERVER PRO SBĚR SENZORICKÝCH DAT A ŘÍZENÍ AKTIVNÍCH PRVKŮ

SERVER FOR COLLECTING SENSOR DATA AND CONTROL OF ACTIVE ELEMENTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JOZEF HALAJ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JAN VIKTORIN**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/20390/2016/xhalaj03

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Halaj Jozef**
Obor: Informační technologie
Téma: **Server pro sběr senzorických dat a řízení aktivních prvků**
Server for Collecting Sensor Data and Control of Active Elements
Kategorie: Informační systémy

Pokyny:

1. Seznamte se s projektem BeeeOn (systém pro řízení inteligentní domácnosti) a s problematikou komunikace se vzdálenými senzory a aktivními prvky.
2. Nastudujte principy komunikace pomocí technologie Web Socket s ohledem na bezpečnost a možnost obsluhovat vysoký počet vzdálených zařízení.
3. Navrhněte princip komunikace mezi serverem a vzdálenými senzorickými a aktivními prvky. Zohledněte zejm. spolehlivost doručení dat, množství různých měřených veličin, testovatelnost a škálovatelnost.
4. Implementujte server využívající navržený princip komunikace.
5. Otestujte řešení pomocí automatizovaných a poloautomatizovaných testů.
6. Zhodnoťte dosažené výsledky a diskutujte další možnosti práce.

Literatura:

- Dle dohody s vedoucím.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Viktorin Jan, Ing.**, UPSY FIT VUT
Datum zadání: 21. června 2017
Datum odevzdání: 31. července 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Bakalárska práca sa zaoberá problematikou komunikácie so vzdialenými senzormi a aktívnymi prvkami v systéme inteligentnej domácnosti s názvom BeeeOn. Práca popisuje systém BeeeOn, jeho jednotlivé časti a princíp pôvodnej nevyhovujúcej komunikácie. Pre komunikáciu s bránami BeeeOn je využitá technológia WebSocket, ktorá umožňuje komunikovať i v sieťach s obmedzeným prístupom na privilegované porty. Implementovaný server je v princípe schopný obsluhovať vysoký počet brán BeeeOn, pomocou ktorých sú vzdialené zariadenia pripojené k systému. Komunikácia je zabezpečená pomocou SSL/TLS, používa potvrdzovacie mechanizmy pre zaručenie spoľahlivosti a je jednoducho rozšíriteľná o ďalšie potrebné správy. Prináša do systému možnosť zasielania asynchrónnych príkazov na bránu BeeeOn a pripojené zariadenia. Server je implementovaný v jazyku C++. Najbežnejšie scenáre komunikácie boli otestované automatickými testami.

Abstract

Bachelor's thesis aims on communication with remote sensors and active elements in the smart home system called BeeeOn. The individual parts of the BeeeOn system are described with respect to the current insufficient communication principle and implementation. For communication with BeeeOn gateways, the WebSocket technology is used. It allows communication on netwoks with a restricted access to privileged ports. The implemented server is in principle capable of serving a high number of BeeeOn gateways that works as a bridge among the server and the remote sensors. The communication is secured with SSL/TLS, it uses confirmation mechanism to guarantee reliability and it is easily extendable by other protocol messages. It enables the system to send asynchronous commands to the BeeeOn gateway and to the connected devices. The server is implemented in C++ language. The most common communication scenarios were tested by automated tests.

Kľúčové slová

inteligentná domácnosť, BeeeOn, WebSocket, POCO, C++, komunikácia, server, Gateway

Keywords

smart home, BeeeOn, WebSocket, POCO, C++, communication, server, Gateway

Citácia

HALAJ, Jozef. *Server pro sběr senzorických dat a řízení aktivních prvků*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Viktorin Jan.

Server pro sběr senzorických dat a řízení aktivních prvků

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Jana Viktorina. Uviedol som všetky literárne zdroje a publikácie, z ktorých som čerpal.

.....
Jozef Halaj
28. júla 2017

Poďakovanie

Rád by som sa poďakoval vedúcemu bakalárskej práce Ing. Janovi Viktorinovi za poskytnuté odborné konzultácie a hlavne konštruktívne pripomienky, či už pri písaní textovej časti, návrhu alebo implementácii. Ďalej by som sa chcel poďakovať celému tímu BeeeOn za výbornú spoluprácu na projekte.

Obsah

Kapitola 1

Úvod

Spolu s technickým pokrokom sa zvyšujú požiadavky ľudí na komfort a čo najkvalitnejší životný štandard. Bežné úkony v domácnosti, ktoré nám ešte donedávna boli prirodzené, nahradzujú elektronické spotrebiče, uľahčujúce a zefektívňujúce beh domácnosti. Internet už dávno neslúži len ako zdroj informácií, ale čím ďalej viac preniká do nášho života a postupne aj do bežných spotrebičov.

Vznikajú inteligentné domácnosti, ktorých cieľom je zvýšiť komfort, efektívnosť, bezpečnosť a taktiež znížiť náklady na prevádzku domácnosti. Spotrebiče je možné ovládať na diaľku a domácnosť je možné monitorovať, a taktiež automatizovať. Príkladom môže byť vypnutie osvetlenia a zamknutie dverí po odchode z domu, regulácia vykurovania na základe snímanej teploty alebo zapnutie ohrevu vody.

Inteligentnej domácnosti sa venuje aj projekt BeeeOn vyvíjaný na Fakulte Informačných technológií v Brne. Projekt sa zameriava na jednoduchosť, a zjednotenie monitorovania a ovládania zariadení od rôznych výrobcov. Zariadenia, obsahujúce senzory na snímanie vlastností prostredia a aktívne prvky na ovládanie domácnosti sa pripájajú na bránu inteligentnej domácnosti BeeeOn, pomocou ktorej komunikujú cez internet so zbytkom systému. Ústredným prvkom systému je server, ktorý zabezpečuje centralizované ukladanie dát z pripojených brán a umožňuje zasielať riadiace príkazy zariadeniam v domácnosti.

Táto práca sa zaoberá problematikou komunikácie serveru so vzdialenými senzormi a aktívnymi prvkami v systéme BeeeOn. Jej cieľom je navrhnúť princíp komunikácie medzi serverom a pripojenými domácnosťami skrz bránu BeeeOn, a následne implementovať server využívajúci navrhnutý princíp komunikácie. Navrhnutý princíp komunikácie využíva technológiu WebSocket, čím sú odstránené pôvodné problémy blokovania komunikácie rôznymi firewallmi v existujúcej sieťovej infraštruktúre. Umožňuje bezpečné pripojenie a v princípe obsluhu veľkého počtu brán, poskytuje spoľahlivosť doručenia prenášaných dát, a je jednoducho rozšíriteľný o ďalšie typy potrebných správ. Prináša do systému možnosť zasielania asynchrónnych príkazov na bránu a k nej pripojených zariadení a následné reportovanie stavu vykonávania týchto príkazov.

Text práce je rozdelený do niekoľkých častí. Prvá časť predstavuje a popisuje systém pre riadenie inteligentnej domácnosti BeeeOn, jeho architektúru a jednotlivé súčasti systému. Taktiež popisuje princíp a nevyhovujúci stav komunikácie so senzorickými zariadeniami v domácnosti pripojenými na bránu BeeeOn. Ďalšia časť sa venuje použitým technológiám výsledného riešenia. Popisuje technológiu WebSocket, knižnicu POCO a jej najdôležitejšie použité nástroje. Návrh novej komunikácie a serveru využívajúceho princíp tejto komunikácie, spolu s jeho implementáciou sú popísané v štvrtej kapitole. Na túto časť nadväzuje posledná kapitola predstavujúca testovanie funkčnosti výsledného riešenia práce.

Kapitola 2

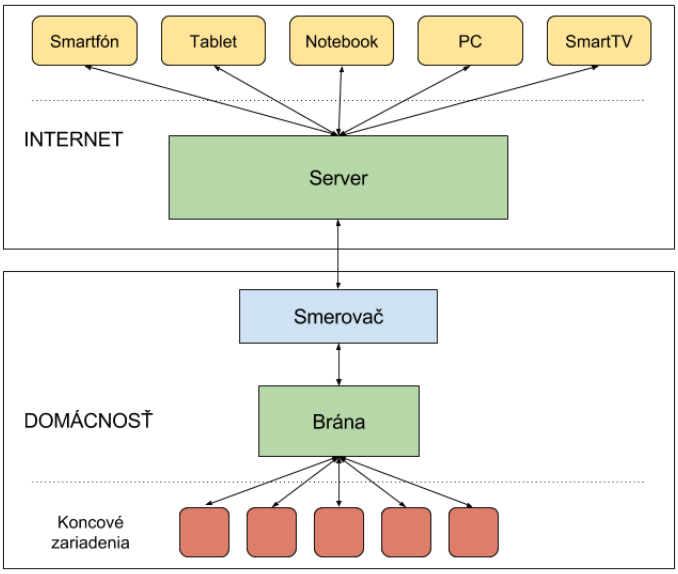
BeeeOn

BeeeOn je open-source projekt vyvíjaný na Fakulte Informačných Technológií VUT v Brne, ktorý sa zaoberá rozvojom Internetu vecí. Jeho cieľom je vytvorenie modulárneho systému pre riadenie a automatizáciu inteligentnej domácnosti, s dôrazom na prístupnosť, bezpečnosť a jednoduchú rozšíriteľnosť o ďalšie prvky.

V súčasnosti systém poskytuje všetky hardvérové komponenty potrebné pre beh inteligentnej domácnosti a rovnako ponúka softvérové riešenia. Technické špecifikácie, detaily návrhu hardvéru, ako aj schémy plošných spojov sú dostupné na oficiálnych stránkach BeeeOn [1] a v repozitároch na serveri GitHub [2], kde sú taktiež zverejňované všetky zdrojové kódy projektu. Nasledujúce informácie o systéme boli čerpané z oficiálnych stránok projektu [1], interných wiki stránok [3][4][6] a zo zdrojových kódov projektu [2].

2.1 Architektúra

Architektúra systému, sa skladá z niekoľkých samostatných komponentov, ktoré sú na najvyššej úrovni abstrakcie zoskupené do štyroch hlavných častí.



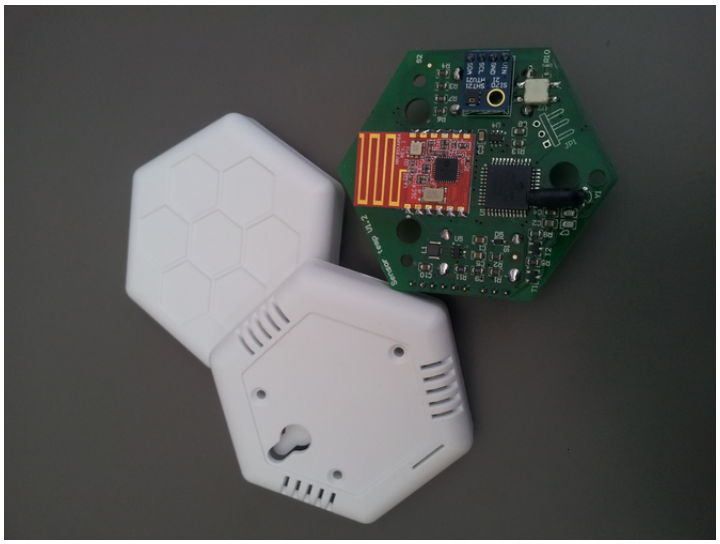
Obr. 2.1: Architektúra systému BeeeOn

Koncové zariadenia vo forme senzorov a aktívnych prvkov sú bezdrôtovo pripojené na bránu. **Brána** predstavuje centrálny prvok domácnosti a spája koncové zariadenia so zbytkom systému. Pomocou smerovača v lokálnej sieti je pripojená do siete Internet, cez ktorú komunikuje so serverom za účelom zasielania dát z koncových zariadení a prijímania riadiacich príkazov. **Server** je zodpovedný za spracovanie požiadaviek a ukladanie dát od brány, ale na druhej strane spracúva tiež požiadavky od užívateľa, ktorý so systémom komunikuje pomocou **užívateľského rozhrania** vo forme mobilnej alebo webovej aplikácie. V súčasnosti BeeeOn poskytuje android aplikáciu, ktorá užívateľovi prezentuje všetky dôležité informácie o jeho bráne a koncových zariadeniach a umožňuje zasielať riadiace príkazy. Webová aplikácia je vo vývoji.

2.2 Koncové zariadenia

Koncové zariadenia sa nachádzajú na spodnej vrstve architektúry. Sú to **bezdrôtovo** pripojené fyzické zariadenia, komunikujúce s centrálnou bránou. Koncovým zariadením môže byť akýkoľvek prístroj v domácnosti, ktorý umožňuje snímať vlastnosti prostredia alebo do prostredia zasahovať. Na základe týchto skutočností ich rozdeľujeme do dvoch skupín:

- **Senzory** slúžia na získavanie dát z okolitého prostredia. Namerané dáta, väčšinou fyzikálne veličiny (napr. teplota, tlak vzduchu, elektrická spotreba atď.), sú v určitých intervaloch zasielané na bránu. Tá ich spracúva a distribuuje do vyšších vrstiev systému.
- **Aktívne prvky (aktuátory)** naopak ovplyvňujú stav okolia na podnet riadiacich pokynov od brány. Môže ísť o jednoduché zopnutie spínača elektrickej zásuvky, ovládanie pohonu termostatickej hlavice a pod.



Obr. 2.2: BeeeOn Sensor [1]

Zariadenia v domácnosti by mali byť primárne jednoduché (jednoúčelové, cenovo dostupné, s nízkymi energetickými nárokmi), ale taktiež to môžu byť komplexné zariadenia, zložené z viacerých senzorov a aktívnych prvkov. Vďaka bezdrôtovej komunikácií s bránou

môžu byť ľubovoľne umiestnené a premiestňované, čo prináša jednoduchú integráciu do každej domácnosti.

V súčastnosti systém podporuje vlastný senzor, vyvinutý v rámci projektu a širokú škálu zariadení tretích strán (napr. Thermona[1], Jablotron[2], Open ZWave[3], Philips Hue[4]). BeeeOn senzor, zobrazený na obrázku 2.2, je schopný merať teplotu (po pripojení externého čidla aj externú), vlhkosť vzduchu a komunikuje pomocou vlastného FIT protokolu. FIT protokol je proprietárny bezdrôtový komunikačný protokol navrhnutý špeciálne pre Internet vecí.

2.2.1 Identifikácia koncových zariadení v systéme

Zariadenia tretích strán komunikujú rôznymi bezdrôtovými protokolmi na rôznych frekvenciách. Preto je nutné, aby bol k bráne rozhraním USB pripojený adaptér, zabezpečujúci komunikáciu s týmito zariadeniami pomocou vhodného protokolu.

Každé koncové zariadenie je v systéme identifikované pomocou jednoznačného identifikátoru. Identifikátor predstavuje 64 bitové číslo a je jedinečný v rámci celého systému BeeeOn. Skladá sa z 8 bitového prefixu, ktorý značí typ komunikačného protokolu a samotnej identifikácie konkrétneho fyzického zariadenia.

Okrem jedinečnej identifikácie konkrétnych zariadení je v systéme potrebné rozpoznávať rôzne typy zariadení. Na rozpoznanie a správu všetkých podporovaných zariadení v systéme BeeeOn slúži **tabuľka zariadení**. Je to XML dokument obsahujúci špecifikácie všetkých podporovaných zariadení. Každé nové podporované zariadenie musí byť pridané do tejto tabuľky, ktorá je zdieľaná medzi jednotlivými časťami systému BeeeOn. Všetky zariadenia v tabuľke sú identifikované jedinečným identifikátorom (*id*) slúžiacim na rozpoznanie typu zariadenia v jednotlivých častiach systému, menom (*name*), výrobcom (*manufacturer*) a modulmi (*modules*). Každý modul v rámci zariadenia má jedinečný identifikátor a môže to byť senzor, aktívny prvok, batéria atď.

```
<device id="29" name="Everspring Wireless SmokeDetector SF812">
        <name>T:DEV_EVERSPRING_SF812_SMOKE_DETECTOR</name>
        <manufacturer>T:MANUFACTURER_EVERSPRING</manufacturer>
        <modules>
                <sensor id="0" type="0x03">
                        <group>T:ZONE_1</group>
                        <name>T:EVERSPRING_SF812_SMOKE_SENSOR</name>
                        <values name="T:EVERSPRING_SF812_SMOKE_SENSOR_STATE">
                                <value id="0">T:OFF</value>
                                <value id="1">T:ON</value>
                        </values>
                </sensor>
                <battery id="1" type="0x08" />
        </modules>
</device>
```

Ukážka 2.1: Príklad špecifikácie zariadenia v tabuľke zariadení

[1] http://www.thermona.cz/
[2] http://www.jablotron.cz/
[3] http://www.openzwave.com/
[4] http://www2.meethue.com/

Aby mohli koncové zariadenia komunikovať so systémom, musia byť spárované s bránou. Po úspešnom spárovaní sú súčasťou systému, teda môžu prijímať príkazy na nastavenie aktívnych prvkov a zasielať dáta bráne.

Spárované zariadenia sa môžu uspať až do času, kedy je naplánované prevedenie ďalšej činnosti – zmeranie meranej veličiny, odoslanie dát alebo prijatie riadiacich pokynov. Tento proces uspávania zvyšuje životnosť batérie. Napríklad teplotný senzor sa môže prebudiť raz za 5 minút, zmeria teplotu, odošle namerané dáta a znova sa uspí. Podobné uspávanie sa môže diať aj u aktívnych prvkov, kde to môže narozdiel od senzorov spôsobiť určité problémy. Napríklad, užívateľ dá pokyn na nastavenie hodnoty daného prvku, ale ten je uspatý a hodnota sa nastaví až po prebudení. Túto skutočnosť je potrebné reflektovať užívateľovi.

2.3 Brána

Brána (historicky taktiež označovaná ako *adaptér*), zobrazená na obrázku 2.3, je fyzické zariadenie predstavujúce centrálny prvok inteligentnej domácnosti BeeeOn. Na jednej strane je pripojená na internet a pomocou zabezpečeného TCP spojenia komunikuje so serverom. Na druhej strane pomocou bezdrôtového protokolu komunikuje s pripojenými zariadeniami v domácnosti.

Obr. 2.3: BeeeOn Gateway [1]

Hlavnou úlohou brány je zabezpečenie obojsmernej komunikácie medzi serverom a koncovými zariadeniami. Zo strany serveru sú prijímané riadiace príkazy, ktoré sa po spracovaní delegujú na koncové prvky. Z druhej strany sú prijímané dáta zo senzorov, ktoré sú po spracovaní bránou odoslané na server.

Brána poskytuje taktiež ochranu proti strate senzorických dát v prípade výpadku internetového spojenia. Počas výpadku sú dáta ukladané najprv do vyrovnávacej pamäte a po dosiahnutí určitej hranice, sú zapisované na SD kartu. Po opätovnom spojení sú všetky uložené dáta postupne poslané na server.

Každá brána v systéme je identifikovaná jednoznačným identifikátorom. Identifikátor predstavuje 16 miestne číslo v dekadickej sústave, ktoré je v rámci systému jedinečné.

Tento identifikátor je vygenerovaný pri výrobe brány spolu s odpovedajúcim QR kódom, ktorý slúži na jednoduchú registráciu brány v mobilnej aplikácií užívateľa.

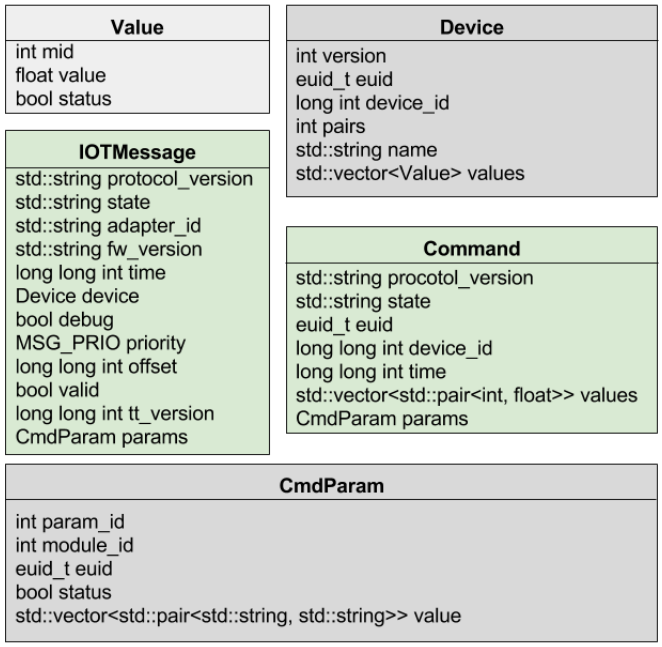
Hardvér

Hardvérovú realizáciu brány predstavuje doska A10-OLinuXino-LIME [18] od spoločnosti Olimex, na ktorej beží operačný systém Linux. Dôležitou súčasťou brány je aj rozširujúca doska, obsahujúca ďalšie potrebné súčasti (napr. rádiovú vrstvu, RTC [5] a vstavaný senzor tlaku)[4]. Práve rádiová vrstva na tejto doske je zodpovedná za komunikáciu s BeeeOn senzormi pomocou FIT protokolu [5]. Pre komunikáciu so zariadeniami tretích strán, ktoré komunikujú rôznymi bezdrôtovými protokolmi, je potrebné pripojiť externý USB adaptér.

2.3.1 Aplikácia AdaApp

Hlavnú funkcionalitu brány zaisťuje aplikácia AdaApp. Okrem obojsmerného prenosu dát medzi serverom a koncovými zariadeniami plní aj ďalšie úlohy. Aplikácia po štarte, a takisto priebežne, kontroluje potrebné prostriedky, ako je nastavený správny čas a zabezpečené pripojenie na server. V prípade, že všetko funguje, prenáša dáta. V opačnom prípade, si ukladá dáta zo senzorickej siete a po obnove prostriedkov ich zasiela na server.

Aplikácia vytvára jednotné rozhranie na komunikáciu medzi serverom a koncovými zariadeniami. Jej návrh bol postavený na modularite jednotlivých častí aplikácie tak, aby bolo možné zapnúť a používať len potrebné moduly aplikácie. Hlavnými a zároveň povinnými modulmi sú agregátor, a modul pre sieťovú komunikáciu, ktoré spolu úzko spolupracujú. Na komunikáciu medzi jednotlivými modulmi slúžia interné štruktúry zobrazené na obrázku 2.4.



Obr. 2.4: Štruktúry na komunikáciu

[5]Real-time clock – hodiny reálneho času

- **Modul pre sieťovú komunikáciu (TCP)** slúži na pripojenie a následnú komunikáciu so serverom. Beží v samostanom vlákne, v ktorom prijíma dáta zo serveru v XML podobe. Po prijatí dát ich prevedie do internej štruktúry `Command` a využitím agregátoru ich deleguje modulom koncových zariadení, pre ktoré sú určené. Naopak agregátor ho využíva na prevod dát z internej štruktúry `IOTMessage` do XML podoby, odoslanie dát na server a prijatie odpovede zo serveru. Komunikácia so serverom je bližšie popísaná v kapitole 2.4.1.

- **Agregátor** je modul tvoriaci centrálny bod komunikácie aplikácie. Beží v samostatnom vlákne, v ktorom sa stará o vyrovnávaciu pamäť a odosielanie správ z tejto pamäte na server. Moduly koncových zariadení ho využívajú na odosielanie dát v štruktúre `IOTMessage`, ktorú predáva sieťovému modulu na odoslanie. Z návratovej hodnoty zistí, či boli dáta skutočne odoslané. V prípade že nie, ukladá ich do vyrovnávacej pamäte. Modul pre sieťovú komunikáciu ho využíva na delegovanie správy v štruktúre `Command` modulom koncových zariadení, pre ktoré je určená.

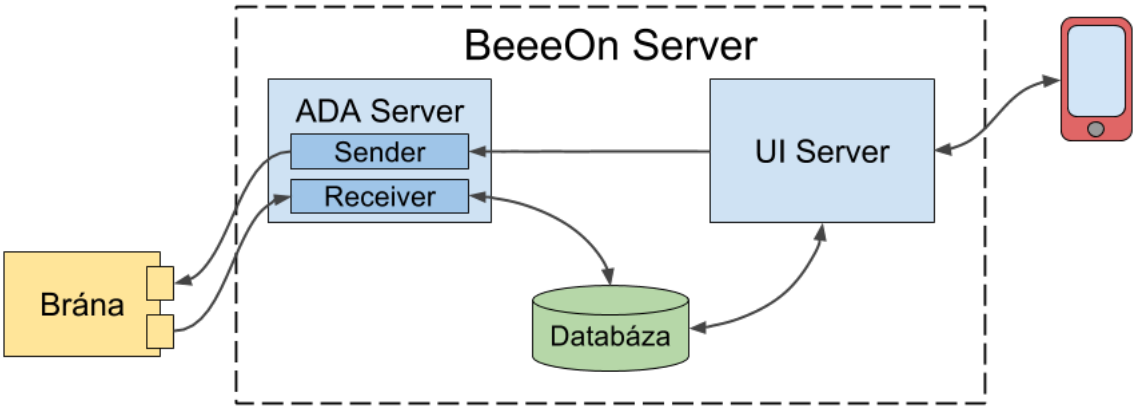
- **Moduly koncových zariadení** zabezpečujú komunikáciu medzi agregátorom a koncovými zariadeniami. Jeden modul obsluhuje sadu zariadení, ktoré využívajú rovnaký komunikačný protokol. Každý modul využíva iný spôsob komunikácie so zariadeniami, ale s agregátorom komunikuje pomocou rozhrania v podobe štruktúr `IOTMessage` a `Command`.

Aplikácia nebola pôvodne stavaná na systematické pridávanie nových typov koncových zariadení. S postupným rozširovaním podpory pre zariadenia tretích strán v systéme bolo nutné zasahovať do jadra aplikácie a upravovať ho. Vytváranie nových modulov pre tieto zariadenia neustále vyžadovalo zmeny v rozhraní vo forme interných štruktúr. Vznikajúce problémy nebolo možné riešiť systematicky a preto sa navrhla nová aplikácia, ktorá sa aktuálne vyvíja spolu s novým serverom.

2.4 Server

Server je ústredným prvkom systému BeeeOn. Zaisťuje hlavnú logiku riadenia inteligentnej domácnosti a je nutný pre komunikáciu medzi užívateľom a koncovými zariadeniami v domácnosti, resp. bránou. Poskytuje dátové úložisko vo forme relačnej databázy, v ktorej sú uložené všetky potrebné informácie o užívateľoch, ich právach, pripojených bránach a koncových zariadeniach, a taktiež je ukladaná história nameraných dát. Je zodpovedný za spracovanie požiadaviek a ukladanie dát od brány, a zároveň prijíma požiadavky od užívateľa a poskytuje mu potrebné informácie alebo zasiela riadiace príkazy na bránu, resp. koncové zariadenia. Server je rozdelený na dve serverové aplikácie, ADA Server a UI Server, ako je zobrazené na obrázku 2.5. Tieto serverové aplikácie sú spustené ako služby na jednom fyzickom serveri s operačným systémom Linux, ale je možné aj ich rozdelenie na rôzne fyzické stroje.

ADA Server je serverová aplikácia zabezpečujúca komunikáciu s pripojenými bránami a ukladanie nameraných dát do databázy. Po pripojení brány si ukladá TCP socket do kontajnera spojení a cez toto spojenie následne zasiela na bránu riadiace príkazy prijímané z UIServera. Pre každú ďalšiu požiadavku a namerané dáta od brány sú vytvárané nové TCP spojenia a taktiež každá správa od UIServera vyžaduje nové TCP spojenie. Tieto dočasné spojenia slúžia len na prijatie správy a zaslanie odpovede a následne sú uzavreté. Aplikácia je rozdelená na dve hlavné časti, ktoré bežia v samostatných vláknach:



Obr. 2.5: BeeeOn Server

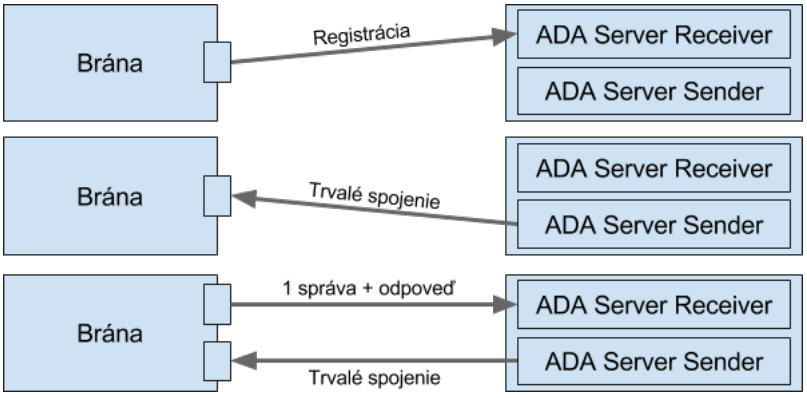
- **ADA Server Receiver** slúži na prijatie prichádzajúceho spojenia s bránou, obsluhu prijatej požiadavky a zaslanie odpovede bráne. V prípade, že sa jedná o registračnú správu, ukladá si spojenie, cez ktoré bude môcť ADA Server Sender zasielať správy bráne.
- **ADA Server Sender** slúži na prijatie prichádzajúceho spojenia s UI Serverom, spracovanie prijatej požiadavky do podoby zrozumiteľnej pre bránu a zaslanie bráne. Na správy zasielané zo serveru brána neodpovedá a preto je UI Serveru zaslaná len informácia, či bola správa odoslaná.

Spracovanie požiadaviek neprebieha vo vláknach týchto dvoch častí, ale využíva sa threadpool (fond vlákien). Vo vláknach threadpoolu sú spúšťané pracovné vlákna, ktoré obslúžia danú požiadavku a ukončia sa. Komunikácia s bránou je bližšie popísaná v kapitole 2.4.1.

UI Server poskytuje užívateľským rozhraniam (v súčasnoti len vo forme mobilnej android aplikácie) možnosť komunikovať so zbytkom systému BeeeOn. Na komunikáciu je využívané TCP spojenie so serverom a proprietárny XML protokol, umožňujúci autentifikáciu užívateľa, ukladanie a získavanie užívateľských dát z databázy a zasielanie riadiacich pokynov na bránu resp. koncové zariadenia.

2.4.1 Komunikácia s bránou

Ako už bolo spomenuté súčasná komunikácia s bránou vyžaduje vytváranie obrovského množstva TCP spojení. Prvé spojenie vytvára brána po zapnutí a cez toto spojenie posiela registračnú správu, na ktorú očakáva odpoveď označujúcu, že server akceptoval jej pripojenie a je schopný s ňou komunikovať. Spojenie je ďalej udržované, za účelom jednosmernej komunikácie zo strany serveru. Zasielanie riadiacich príkazov od užívateľa však vyžaduje vytváranie nového spojenia pre každú správu medzi UI Serverom a ADA Serverom. Na správy iniciované zo strany serveru nechodia od brány žiadne odpovede. Užívateľ sa teda nie je schopný dozvedieť, či bola správa bránou prijatá.



Obr. 2.6: Komunikácia brány so serverom

Správy od brány ako zasielanie dát alebo získanie dodatočných informácií zo strany serveru vyžadujú vytváranie nového spojenia s ADA Serverom pre každú správu. Na tieto správy server zasiela odpoveď cez vytvorené spojenie, ktorým prišla požiadavka. Obrázok 2.6 zobrazuje popísanú komunikáciu medzi bránou a serverom. Správy medzi bránou a serverom majú XML podobu a taktiež medzi UI Serverom a ADA Serverom.

Typy prenášaných správ

Správy zasielané z brány:

- **register** – Požiadavka na registráciu brány na serveri. Správa sa posiela len raz po zapnutí AdaApp.
  - **register** – Potvrdenie od serveru, že je brána zaregistrovaná a je pripravený od nej prijímať dáta. Server si ukladá socket, cez ktorý bude zasielať požiadavky na bránu.
- **data** – Správa slúži na prenos dát z koncových zariadení. Zariadenia posielajú naraz celú svoju štruktúru hodnôt zo všetkých senzorov a taktiež aktívnych prvkov.
  - **update** – Potvrdenie od serveru, že prijal nové dáta. Súčasťou správy je položka `time`, ktorá slúži na nastavenie intervalu pre získavanie hodnôt z koncových prvkov.

Správy zasielané zo servera:

- **set** – Nastavenie aktívneho prvku na novú hodnotu.
- **listen** – Prepnutie brány do párovacieho režimu.
- **clean** – Odpárovanie koncového zariadenia.

Pomocou spomínaných správ nebolo možné prenášať všetky potrebné informácie medzi bránou a servervom, preto bol XML protokol rozšírený o ďalšie typy správ:

- **getparameters** – Požiadavka brány na získanie dodatočných informácií. Obsahuje kód požiadavky, ktorý označuje typ požadovaných dát. Napríklad, môže ísť o získanie poslednej nameranej hodnoty nejakého modulu zariadenia alebo získanie spárovaných zariadení.
- **parameters** – Odpoveď od serveru na predchádzajúcu správu.

2.4.2 Nový server

Popísaný server je produkčne nasadený, udržiava sa, ale v súčasnosti sa vyvíja nový server kompletne prepísaný nad knižnicou POCO. Dôvodov pre vznik nového serveru bolo viacero. Jedným z nich bol postupný prechod z proprietárneho XML protokolu na REST API, kvôli rozširovaniu užívateľského rozhrania o webovú aplikáciu. Pôvodný server na to nebol stavaný a bolo komplikované v ňom robiť zmeny. Ďalším dôvodom bolo, že jednotlivé časti serveru implementovali rovnaké alebo podobné problémy rôznymi spôsobmi, využívali mix rôznych technológií alebo nevyužívali dostupné technológie vôbec a implementovali problémy vlastným spôsobom. V jednotlivých častiach serveru vznikali zložité závislosti a preto nebolo možné robiť jednoduché zmeny a systém rozširovať.

Novo vyvíjaný server BeeeOn [7] využíva Layered (vrstvovú) architektúru [20]. Niekoľko vrstiev umožňuje oddelenie zodpovednosti, uľahčenie testovania a taktiež možnosť zmeny niektorej vrstvy bez zásadného vplyvu na vrstvy ostatné (napr. zmena databázového backendu, zmena API, atď.). Rozlišujeme 3 základné vrstvy:

- **Prezentačná vrstva** realizuje komunikáciu s bránami, mobilmi, apod.
- **Biznis vrstva (Business)** obsahuje vlastnú logiku serveru.
- **Dátová vrstva (DAO)** realizuje prístup k dátam v databáze.

Všetky vrstvy sú previazané triedami z dátového modelu. Typicky platí, že biznis vrstva väčšinou nepozná detaily ani vrstvy prezentačnej ani dátovej. Prakticky teda nedokáže konštruovať objekty dátového modelu, ale ich inštancie dostáva od vrstiev susedných.

Všetky časti serveru používajú spoločný dátový model systému. Môže ísť o objekty User, Gateway, Device, DeviceInfo, ModuleInfo atď.

Do dátovej vrstvy okrem prístupu k databáze patria aj poskytovatelia (providers). Provider poskytuje implementáciu nejakej operácie pre zbytok systému. Môže to byť napr. RandomProvider poskytujúci náhodné čísla alebo DeviceInfoProvider poskytujúci informácie o zariadení (prístup k devices.xml), atď.

Server pozostáva z veľkého množstva tried, ktoré je potrebné vhodne nakonfigurovať a poskladať dohromady. Pretože sa nejedná o jednoduchý úkol vzhľadom na množstvo závislostí je runtime serveru zostavovaný podľa princípu Dependency Injection [10], na základe konfiguračného súboru. Jednotlivým objektom sú závislosti injektované a využitím polymorfizmu je takto možné jednoducho vymeniť implementáciu niektorej časti systému.

V súčasnosti je implementovaný kompletný dátový model spolu s dátovou vrstvou. Nad ňou je implementovaná serverová časť UI Server s jej biznis vrstvou a prezentačnou vrstvou s proprietárnym XML protokolom. Nový UI Server je schopný komunikovať s pôvodným ADA Serverom a postupne sa vyvíja prezentačná vrstva v podobe REST API.

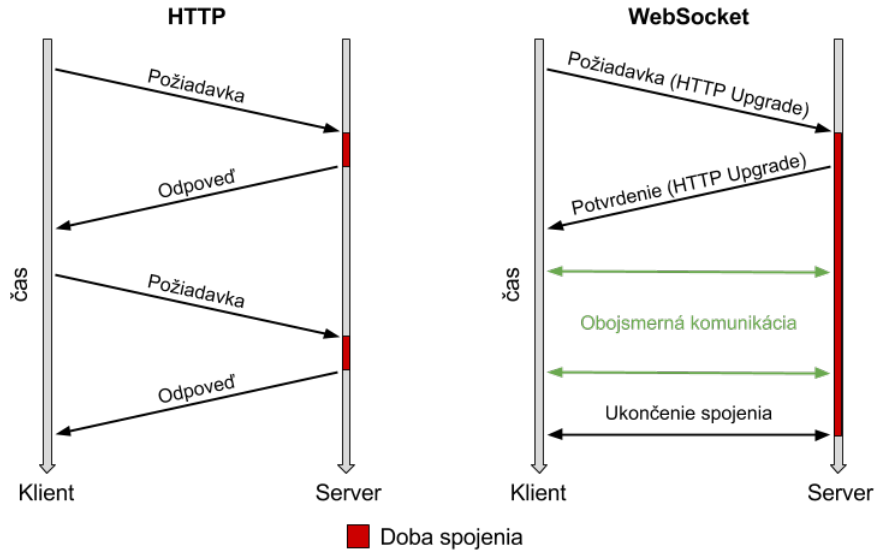
Kapitola 3

Použité technológie

3.1 WebSocket

WebSocket je aplikačný protokol poskytujúci full-duplex komunikáciu skrz jedno TCP spojenie. Protokol bol štandardizovaný v roku 2011 a definovaný v RFC 6455 s názvom The WebSocket protocol[9]. Primárne bol navrhnutý pre použitie vo webových aplikáciách vyžadujúcich obojsmernú komunikáciu v reálnom čase (napr. chat alebo herné aplikácie) medzi webovým prehliadačom a serverom, ale je použiteľný aj v iných aplikáciách typu klient – server. Uľahčuje prenos dát v reálnom čase zo servera a na server, čím prináša viac interakcie medzi prehliadačom a webovým serverom. Toto je možné tým, že poskytuje serveru štandardizovanú cestu na zaslanie dát prehliadaču bez toho, aby si to vyžiadal klient a umožňuje, aby boli správy prenášané tam a späť pri zachovaní otvoreného spojenia.

WebSocket je nezávislý protokol založený na TCP, avšak pre ustálenie spojenia (handshake) na prenos dát využíva HTTP[1] protokol.



Obr. 3.1: Porovnanie HTTP a WebSocket komunikácie

[1]Hypertextový prenosový protokol

Obrázok 3.1 zobrazuje porovnanie komunikácie medzi klientom a serverom u protokolov HTTP a WebSocket. Zatiaľ čo HTTP pracuje formou dotaz/odpoveď zo strany klienta a klient pre každú požiadavku typicky vytvára nové spojenie, WebSocket po ustálení spojenia umožňuje plne duplexnú komunikáciu z oboch smerov a až do požiadavky na ukončenie spojenia, zachováva spojenie otvorené.

Z historického hľadiska, vytváranie webových aplikácií, ktoré potrebujú obojsmernú komunikáciu v reálnom čase medzi klientom a serverom, vyžadovalo zneužitie HTTP na dotazovanie servera na dáta, zatiaľ čo zasielanie správ z klienta bolo realizované novými spojeniami. Toto provizórne riešenie má za následok rôzne problémy:

- Server je nútený používať množstvo rôznych TCP spojení pre každého klienta. Jedno spojenie na zasielanie dát klientovi a ďalšie pre každú prichádzajúcu správu.
- Vysoká réžia, spôsobená HTTP hlavičkou v každej správe a vytváraním nových TCP spojení.

Jednoduchšie a lepšie riešenie je použitie jedného TCP spojenia pre obojsmernú komunikáciu, čo ponúka WebSocket. Je navrhnutý pre nahradenie existujúcich neštandardizovaných obojsmerných komunikačných technológii (napr. Comet), ktoré používajú HTTP na transport kôli ťaženiu z existujúcej sieťovej infraštruktúry (proxy servery, firewall, filtrovanie, autentizácia). V kontexte využitia existujúcej webovej infraštruktúry, nasleduje existujúce technológie. Pracuje na HTTP porte 80 resp. 443 pre zabezpečenú komunikáciu pomocou SSL/TLS a taktiež podporuje HTTP proxy servery, firewall, atď. Štandard však uvádza, že WebSocket nie je viazaný na HTTP a budúce implementácie by mohli využiť jednoduchší handshake cez dedikovaný port bez nutnosti zmeny celého protokolu.

Špecifikácia WebSocket protokolu definuje `ws` a `wss` ako 2 nové URI[2] adresy pre nezabezpečené a zabezpečené spojenie v nasledujúcom formáte:

```
ws-URI = "ws://"host [ ":"port ] path [ "?"query ]
wss-URI = "wss://"host [ ":"port ] path [ "?"query ]
```

Východzia hodnota portu je 80 pre `ws` a 443 pre `wss`.

WebSocket protokol pozostáva z dvoch častí, konkrétne handshake a prenos dát, ktoré sú popísané v nasledujúcich kapitolách.

3.1.1 Handshake

Na nadviazanie WebSocket spojenia klient zasiela WebSocket handshake požiadavku. Jedná sa o HTTP Upgrade požiadavku z dôvodu spomínanej kompatibility. Výhodou je, že jeden port môže byť použitý ako pre HTTP, tak pre WebSocket. Server mu odpovedá Websocket handshake odpoveďou. Po úspešnom nadviazaní spojenia je vytvorený obojsmerný komunikačný kanál, kde každá strana môže zasielať dáta nezávisle na druhej strane.

Obrázok 3.2 zobrazuje príklad handshake požiadavky. Prvý riadok indikuje že sa jedná o HTTP hlavičku a povinne obsahuje metódu GET. URI adresa tejto metódy identifikuje koncový bod WebSocket spojenia, čo umožňuje, aby na jednom serveri existovalo niekoľko rôznych koncových bodov. Položka `Connection: Upgrade` v kombinácií s položkou `Upgrade: websocket` označujú, že klient chce spojenie povýšiť na WebSocket. `Host` slúži na potvrdenie názvu hostiteľa danej služby a v položke `Origin` je predaný kontext použitia.

[2] Universal Resource Identifier - jednotný identifikátor zdroja

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Obr. 3.2: WebSocket handshake požiadavka

`Sec-WebSocket-Key` je náhodne vygenerovaný base64 reťazec použitý na autentizáciu servera. Server klientovi vráti hodnotu zostavenú z 2 kľúčov. Prvým kľúčom je práve hodnota položky `Sec-WebSocket-Key` odoslanej klientom. Táto hodnota je zbavená všetkých bielych znakov pred a za textom, následne je spojená s GUID[3] pre WebSocket protokol. Spojenie týchto kľúčov je odoslané klientovi ako hash hodnota vytvorená pomocou SHA-1[4]. Klient obdrží túto hodnotu v base64 kódovaní v položke `Sec-WebSocket-Accept`, čím si overí, že skutočne odpovedá server, ktorý požiadavku prijal. Položka `Sec-WebSocket-Protocol` slúži na označenie aké subprotokoly (protokoly aplikačnej vrstvy zabalené do WebSocketu) klient podporuje. Server si vyberie jeden alebo žiadny z týchto protokolov a vráti túto hodnotu klientovi v svojej handshake odpovedi. Klient z položky hlavičky rovnakého názvu zistí, ktorý subprotokol server zvolil. Posledná položka `Sec-WebSocket-Version` označuje verziu WebSocket protokolu. V súčasnosti je štandardizovaná hodnota 13. V prípade, že server nepodporuje danú verziu protokolu, môže s klientom vyjednávať o použití inej zaslaním podporujúcich verzií v handshake odpovedi.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Obr. 3.3: WebSocket handshake odpoveď

Obrázok 3.3 zobrazuje odpoveď od servera. Prvý riadok obsahuje stavový kód `101 Switching Protocols`. Každý iný stavový kód ako 101 značí, že handshake neprebehol úspešne. Po úspešnom handshaku je vytvorené WebSocket spojenie a nasleduje prenos dát.

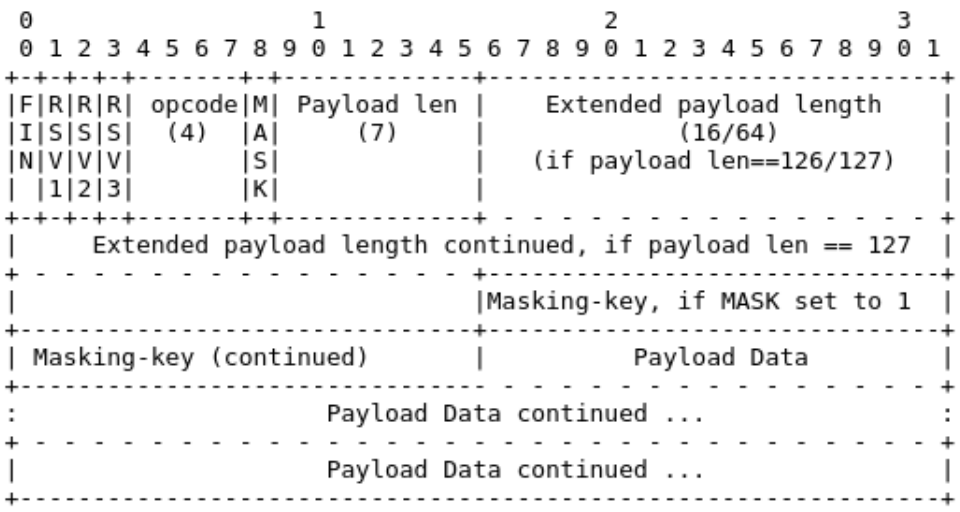
3.1.2 Prenos dát

WebSocket protokol prenáša dáta použitím sekvencie rámcov. Prenosy dát sú nazývané správy, kde jedna správa môže byť rozdelená (fragmentovaná) do niekoľkých rámcov. Fragmentácia dovoľuje odosielať správy s neznámou veľkosťou v prípade odosielania bez využitia vyrovnávacej pamäti. Protokol umožňuje prenos textových dát v kódovaní UTF-8 a taktiež binárnych dát. Definuje rámec pomocou operačného kódu, veľkosti užitočných dát a samotných užitočných dát (payload). Niektoré bity a operačné kódy rámca sú rezervované pre budúcu expanziu protokolu.

Z bezpečnostných dôvodov musí klientská implementácia protokolu maskovať všetky dáta, ktoré zasiela na server. Dáta sú maskované binárnou operáciou XOR pomocou ná-

[3] Globally Unique Identifier
[4] Secure Hash Algorithm

hodne vygenerovaného 32 bitového kľúča, ktorý je pridaný do hlavičky rámca. Maskovanie nijako nezabezpečuje prenášané dáta, keďže kľúč je zasielaný spolu s dátami, ale slúži na zabezpečenie náchylnej sieťovej architektúry pred potenciálnym útokom z klientskej aplikácie (proxy cache poisoning). Nevýhodou je, že protokol vyžaduje maskovanie aj v prípade zabezpečenej komunikácie WSS, kedy je maskovanie nadbytočné.



Obr. 3.4: Dátový rámec WebSocketu [9]

Obrázok 3.4 znázorňuje jednoduchú štruktúru rámca. Prvý bit FIN označuje, či sa jedná o finálny rámec správy. V prípade zasielania nefragmentovaných správ je prvý rámec zároveň posledným. Nasledujúce 3 bity RSV1, RSV2, RSV3 sú rezervované pre rozšírenia protokolu. Položka opcode definuje typ rámca a ako interpretovať užitočné dáta (payload). Typy rámcov s hodnotami opcode v hexadecimálnej reprezentácií:

- 0x0 - nadväzujúci dátový rámec obsahujúci dáta, ktoré nasledujú bezprostredne za predchádzajúcim rámcom v prípade fragmentácie.
- 0x1 - textový dátový rámec obsahujúci dáta v kódovaní UTF-8.
- 0x2 - binárny dátový rámec obsahujúci binárne dáta.
- 0x3-7 - hodnoty rezervované pre budúce dátové rámce.
- 0x8 - riadiaci rámec uzatvorenie WebSocket spojenia.
- 0x9 - riadiaci rámec PING slúži k overeniu živosti spojenia.
- 0x1 - riadiaci rámec PONG slúži ako reakcia na prijatie PING rámca.
- 0xB-F - hodnoty rezervované pre budúce riadiace rámce.

Ďalší bit MASK definuje, či sú užitočné dáta rámca maskované. Maskované sú všetky rámce zasielané od klienta a obsahujú 32 bitový kľúč v položke masking-key, ktorý server použije na odmaskovanie dát. Server dáta nikdy nemaskuje. Takže rámec odoslaný zo servera masking-key neobsahuje.

Veľkosť užitočných dát (položka payload length) v bytoch je reprezentovaná pomocou 7, 7 + 16 alebo 7 + 64 bitov. V prípade, že hodnota na prvých siedmych bitoch je v rozsahu 0 až 125 táto hodnota určuje veľkosť užitočných dát. Hodnoty 126 a 127 určujú, že veľkosť reprezentuje nasledujúcich 16 resp. 64 bitov. Veľkosť dát je interpretovaná ako unsigned integer preto v prípade 64 bitov musí byť hodnota najviac významového bitu 0.

Tabuľka 3.1: Réžia WebSocket protokolu

Payload	**Réžia[Byte]**	
[Byte]	**Klient–Server**	**Server–Klient**
< 126	6	2
$< 2\hat{}16$	8	4
$< 2\hat{}63$	12	8

Tabuľka 3.1 zobrazuje réžiu WebSocket protokolu oproti TCP pri rôznej veľkosti prenášaných dát. Z dôvodu maskovania dát zo strany klienta je réžia väčšia práve o maskovací kľúč. Tabuľka ukazuje, že réžia je oproti prenášaným dátam minimálna, čo je veľkou výhodou WebSocket protokolu.

3.1.3 Zabezpečenie komunikácie

Ako už bolo spomenuté maskovanie nijako nezabezpečuje komunikáciu pomocou WebSocket protokolu. Pre zabezpečenie komunikácie je potrebné využitie WSS a teda tunelovanie cez SSL/TLS. TLS (Transport Layer Security) a jeho predchodca SSL (Secure Sockets Layer) sú šifrovacie protokoly používané na zabezpečenie sieťovej komunikácie. Pomocou kryptografie umožňujú aplikáciam komunikovať po sieti spôsobom, ktorý zabraňuje odpočúvaniu či falšovaniu správ a poskytujú koncovým bodom autentizáciu.

Pre ustavenie zabezpečeného spojenia je využitá asymetrická kryptografia a teda použitie dvojice kľúčov – verejného a súkromného. Klient zasiela požiadavku na vytvorenie zabezpečeného spojenia spolu s rôznymi parametrami spojenia. Server spolu s odpoveďou zasiela svoj certifikát, ktorý okrem iného obsahuje verejný kľúč. Zároveň si môže vyžiadať certifikát od klienta pre overenie jeho identity. Klient si overí identitu servera, vygeneruje základ šifrovacieho kľúča, ktorým sa bude šifrovať následná komunikáciu, zašifruje ho verejným kľúčom servera a pošle mu ho. Server použije svoj súkromný kľúč k rozšifrovaniu základu šifrovacieho kľúča a následne server aj klient z tohto základu a dohodnutých parametrov spojenia vygenerujú hlavný šifrovací kód. Nakoniec si navzájom overia a potvrdia, že komunikáciu bude šifrovaná týmto kľúčom a fáza handshake končí. Následná komunikácia je zabezpečená pomocou symetrickej kryptografie s využítím vytvoreného hlavného šifrovacieho kľúča [19].

Typicky, napr. pri komunikácií s webovými servermi je autentizovaný len server, zatiaľ čo klient zostáva neautentizovaný. To znamená, že totožnosť servera je overená a webový prehliadač si môže byť istý s kým komunikuje.

Pre zabezpečenie komunikácie medzi BeeeOn bránou a serverom je potrebná vzájomná autentizácia. Brána potrebuje mať zaručené, že komunikuje so správnym serverom a takisto je potrebná autorizácia brány a overenie jej identity z certifikátu.

3.1.4 Zhrnutie

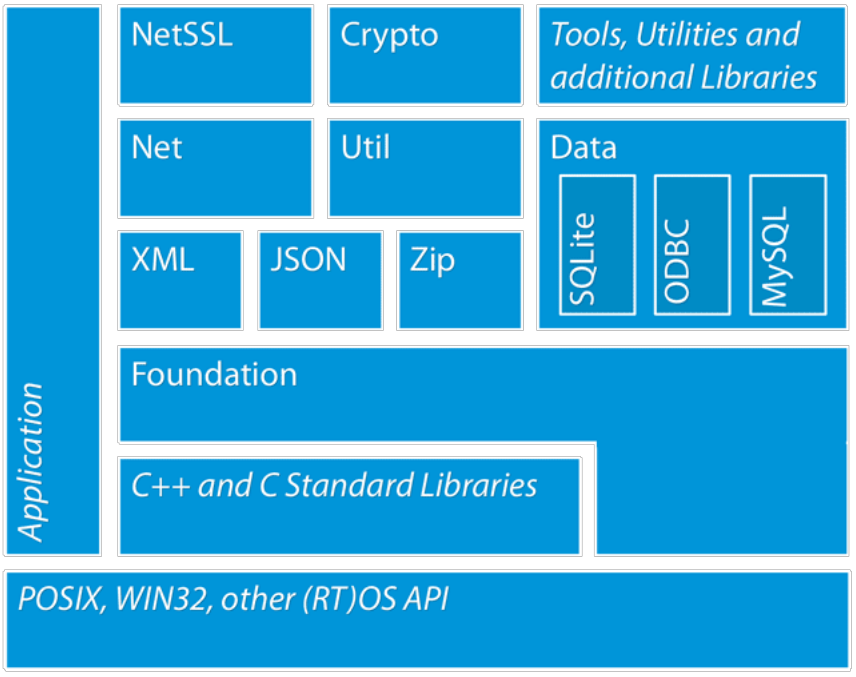
WebSocket je aplikačný protokol využívajúci transportný protokol TCP. Poskytuje perzistentné full-duplex spojenie medzi klientom a serverom. Využíva privilegované porty HTTP a spojenie je vytvárané pomocou HTTP Upgrade požiadavky vďaka čomu dokáže prejsť firewallmi existujúcej sieťovej infraštruktúry. Bol vyvinutý primárne pre použitie v real-time webových aplikáciách, ale je vhodný aj na použitie v iných klient-server aplikáciách. Dáta sú prenášané formou správ (rámcov), ktoré môžu byť fragmentované.

Vďaka tomu, že WebSocket poskytuje zabezpečenú real-time komunikáciu, nízku réžiu prenášaných dát a nemá problémy s existujúcou sieťovou infraštruktúrou je vhodný na použitie v systéme BeeeOn pre komunikáciu medzi serverom a vzdialenými bránami v domácnostiach.

3.2 Knižnica POCO

Keďže štandardná knižnica C++ neobsahuje všetky potrebné funkcie a nástroje na vývoj, je potrebné využitie externých knižníc pre uľahčenie a urýchlenie práce. V projekte BeeeOn sa využíva hlavne POCO[5] a to takmer na úrovni štandardnej knižnice C++.

POCO (POrtable COmponents) je sada open–source knižníc, ktoré zjednodušujú a urýchľujú vývoj sieťovo zameraných, prenosných aplikácií v jazyku C++ [11]. Knižnice sa integrujú so štandardnou knižnicou C++ a zapĺňajú mnohé funkčné medzery. Ich modulárny a efektívny návrh a implementácia robia POCO C++ knižnice vhodné pre vstavaný vývoj, oblasť, kde sa programovací jazyk C++ stáva čoraz obľúbenejší, pretože je vhodný pre nízke úrovne (I/O zariadenia, obsluha prerušení, atď.) a na vysokej úrovni objektovo-orientovaný vývoj.



Obr. 3.5: Architektúra POCO [13]

[5] https://pocoproject.org/

V súčasnosti sú POCO knižnice dostupné v dvoch balíkoch [12]. Základná edícia nevyžaduje externé závislosti a obsahuje knižnice Foundation, Util, XML, JSON a Net. Kompletná edícia obsahuje ďalšie knižnice napr. NetSSL alebo Data, ktoré však vyžadujú externé závislosti (OpenSSL[6], MySQL[7], atď). Prehľad vybraných knižníc POCO:

- **Foundation** – Srdce knižníc POCO. Zahŕňa abstraktnú vrstvu nad platformou ako aj najpoužívanejšie triedy a funkcie. Môže ísť o rôzne dátové typy, funkcie na ich prevody, pokročilejšiu prácu s reťazcami, ktorá v štandardnej knižnici C++ chýba alebo prácu s časom a dátumom.

  Ponúka mnôžstvo tried na správu pamäti, vrátane rôznych chytrých ukazateľov a počítania referencií. Ďalej umožňuje lepšie odchytávanie chýb v podobe rôznych rozšírených výnimiek a ponúka komplexný framework na logovanie.

  Základná knižnica obsahuje aj multivláknovú podporu v podobe POCO vlákien, fondu vlákien, či rôznych synchronizačných prostriedkov. V mnohých aplikáciách potrebujú jednotlivé časti navzájom komunikovať a k tomu v POCO slúžia notifikácie, eventy, centrum notifikácií alebo fronta notifikácií.

- **Util** – Obsahuje framework pre vytváranie konzolových a serverových aplikácií. Zahrnuté je spracovanie argumentov príkazového riadku a manažovanie konfiguračných informácií.

- **XML** – Zastrešuje nástroje pre čítanie, spracovanie a zápis jazyka XML. Podporuje SAX[8] (verzia 2) a DOM[9].

- **JSON** – Umožňuje jednoduchú prácu so serializačným formátom JSON, jeho uchovanie v podobe JSON objektu, parsovanie a taktiež vytváranie.

- **Net** – Knižnica určená na sieťovú komunikáciu. Obsahuje triedy socketov (napr. TCP stream, UDP, WebSocket atď.), TCP server framework, reactor server framework, HTTP klient a server framework a mnoho iných nástrojov pre sieťovú komunikáciu.

- **NetSSL** – Rozširuje knižnicu Net o podporu šifrovania komunikácie pomocou SSL/TLS.

- **Crypto** – Podpora kryptografie.

- **Data** – Poskytuje databázovú abstraktnú vrstvu pre prácu s rôznymi SQL databázami. V súčasnosti sú podporované databázové konektory SQLite, MySQL a ODBC.

Nasledujúce kapitoly popisujú niektoré významné nástroje knižnice POCO, použité vo výslednom riešení práce.

3.2.1 Multithreading

V serverovej aplikácií je potreba vykonávavať rôzne úlohy paralelne. K tomuto účelu sa využívajú procesy a vlákna. POCO poskytuje skupinu tried na paralelné vykonávanie úloh [14].

[6] https://www.openssl.org/
[7] https://www.mysql.com/
[8] Simple Api for XML
[9] Document Object Model

Trieda `Poco::Thread`[10] reprezentuje vlákno a obaľuje systémové vlákna podľa platformy. Vstupným bodom pre vlákno je rozhranie triedy `Poco::Runnable`[11]. Podtriedy musia prepísať metódu `run()`, ktorá sa následne spustí vo vlákne. Taktiež je možné vo vlákne spustiť `Poco::RunnableAdapter`[12] (špecifikuje triednu metódu bez návratovej hodnoty a parametrov) alebo lambda funkciu.

Vytvorenie nového vlákna vtrvá určitý čas. Vlákna môžu byť často znovu použiteľné a taktiež manažovanie životného cyklu vlákna môže byť komplikované. K tomuto účelu slúži `Poco::ThreadPool`[13] (fond vlákien). Threadpool udržuje niekoľko vlákien stále alokovaných a pripravených na spustenie úlohy. Vlákna, ktoré dokončili vykonávanie úlohy sú znovu využité. Threadpoolu je možné nastaviť minimálnu a maximálnu kapacitu. Minimálna kapacita určuje počet vždy alokovaných vlákien. Maximálna kapacita zase určuje počet vlákien, ktoré sa celkovo môžu naalokovať. V prípade dosiahnutia maximálnej kapacity je vyhodená výnimka, že aktuálne nie je dostupné žiadne vlákno. Nečinné vlákna sú po nastaviteľnom časovom limite zničené až do minimálnej kapacity.

Vlákna zdieľajú spoločný pamäťový priestor, preto je pri vykonávaní paralelných úloh potrebná synchronizácia. POCO poskytuje niekoľko synchronizačných primitív. Najpoužívanejšími sú `Poco::Mutex`[14] (rekurzívny zámok) a `Poco::FastMutex`[15] (nerekurzívny zámok). Veľmi užitočným a potrebným je `Poco::ScopedLock`[16], objekt ktorý zamkne zámok pri svojej konštrukcií a pri deštrukcií ho odomkne. Tým sa zamedzí uviaznutiu v prípade, že by bola napr. vyhodená výnimka v kritickej sekcií a zámok by sa neodomkol.

Poco::Util::Timer

`Poco::Util::Timer` [17] umožňuje naplánovanie úloh pre budúce vykonanie vo vlákne časovača na pozadí. Úlohy môžu byť naplánované na jednorazové alebo pravidelné vykonávanie v pravidelných intervaloch.

Časovač vytvorí vlákno, ktoré sekvenčne vykoná všetky naplánované úlohy. Úlohy by mali preto čo najrýchlejšie dokončiť svoju prácu, inak môžu byť ďalšie úlohy oneskorené. Časovač je určený na viacvláknové použitie, teda viaceré vlákna môžu naplánovať nové úlohy súčasne. Úlohu predstavuje objekt, dediaci z `Poco::Util::TimerTask`[17], prepisujúci metódu `run()`. Jednotlivé úlohy môžu byť pred vykonaním zrušené.

3.2.2 JSON

Pre prácu s formátom JSON v POCO slúži knižnica JSON zložená z niekoľkých tried. Trieda `Poco::JSON::Object`[18] reprezentuje objekt JSON a poskytuje reprezentáciu založenú na chytrých ukazateľoch optimalizovanú pre výkon. Pomocou triednych metód je možné jednoducho vytvoriť reprezentáciu požadovaného objektu JSON a taktiež extrahovať hodnoty atribútov alebo previesť celý objekt do textovej podoby. Na parsovanie objektu JSON

[10] https://pocoproject.org/docs/Poco.Thread.html
[11] https://pocoproject.org/docs/Poco.Runnable.html
[12] https://pocoproject.org/docs/Poco.RunnableAdapter.html
[13] https://pocoproject.org/docs/Poco.ThreadPool.html
[14] https://pocoproject.org/docs/Poco.Mutex.html
[15] https://www.appinf.com/docs/poco/Poco.FastMutex.html
[16] https://pocoproject.org/docs/Poco.ScopedLock.html
[17] https://pocoproject.org/docs/Poco.Util.TimerTask.html
[18] https://pocoproject.org/docs/Poco.JSON.Object.html

z textovej podoby slúži `Poco::JSON::Parser`[19]. `Poco::JSON::PrintHandler`[20] umožňuje priame vytváranie a výpis objektu JSON na predaný `std::ostream`.

3.2.3 Network Programming

Pre sieťové programovanie slúži knižnica Net [15]. Poskytuje hierarchiu tried socketov, využívajúcich BSD sockety a taktiež ich zabezpečené varianty pomocou SSL/TLS.

`Poco::Net::StreamSocket`[21] je používaný na klientskej strane na vytvorenie TCP spojenia na server, `Poco::Net::ServerSocket`[22] zas na vytvorenie TCP serverového socketu. Špeciálnym prípadom je `Poco::Net::WebSocket`[23], ktorý je využívaný aj na klientskej aj serverovej strane.

Serverová strana vyžaduje použitie frameworku `Poco::Net::HTTPServer`[24], ktorý poskytuje viacvláknový HTTP server. Využíva `Poco::Net::ServerSocket` a triedu zdedenú od `Poco::Net::HTTPRequestHandlerFactory` (továreň). Táto továreň potom podľa HTTP požiadavky vytvára triedy dediace od `Poco::Net::HTTPRequestHandler`[25], v ktorých sa požiadavka obslúži. V prípade WebSocketu sa v tejto obslužnej triede len jednoducho z objektov `Poco::Net::HTTPServerRequest`[26] a `Poco::Net::HTTPServerResponse`[27] skonštruuje WebSocket.

Poco::Net::SocketReactor

`Poco::Net::SocketReactor` [16] predstavuje časť Initiation Dispatcher (iniciačný dispečer) v návrhovom vzore Reactor [8]. Návrhový vzor Reactor spracováva požiadavky na služby, ktoré sú doručované konkurentne aplikácií jedným alebo viacerými klientmi. Každá služba v aplikácií môže pozostávať z niekoľkým metód a predstavuje samostatnú implementáciu obsluhy udalosti. Obsluha udalosti je zodpovedná za obsluhu špecifických servisných požiadaviek. Za vyvolanie obsluhy udalosti je zodpovedný Initiation Dispatcher, ktorý manažuje registrované obsluhy udalostí. Demultiplexovanie žiadostí o služby je vykonávané synchrónnym demultiplexorom udalostí.

Obsluha udalosti (akákoľvek trieda) môže byť zaregistrovaná a odregistrovaná pomocou metód `addEventHandler()` resp. `removeEventHandler()`. Vždy je registrovaná pre určitý socket predaný ako parameter, spolu s ľubovoľnou metódou, ktorej parametrom je inštancia triedy `Poco::Net::SocketNotification` alebo jej podtriedy.

SocketReactor po štarte čaká na udalosti na registrovaných socketoch, využitím volania `Poco::Net::Socket::select()`[28]. Táto metóda podľa dostupnosti využíva systémové volania *epoll, poll* alebo *select*, čím dokáže sledovať zmenu stavu na veľkom množstve socketov. V prípade zistenia udalosti, je vyvolaná príslušná obsluha udalosti. Existujú 3 udalosti na socketoch a odpovedajúce notifikácie, ktoré vyvolajú obsluhu udalosti:

- ReadableNotification – Zo socketu je možné čítať dáta alebo bolo spojenie uzavreté.

[19] https://pocoproject.org/docs/Poco.JSON.Parser.html
[20] https://pocoproject.org/docs/Poco.JSON.PrintHandler.html
[21] https://www.appinf.com/docs/poco/Poco.Net.StreamSocket.html
[22] https://pocoproject.org/docs/Poco.Net.ServerSocket.html
[23] https://pocoproject.org/docs/Poco.Net.WebSocket.html
[24] https://pocoproject.org/docs/Poco.Net.HTTPServer.html
[25] https://pocoproject.org/docs/Poco.Net.HTTPRequestHandler.html
[26] https://pocoproject.org/docs/Poco.Net.HTTPServerRequest.html
[27] https://www.appinf.com/docs/poco/Poco.Net.HTTPServerResponse.html
[28] https://pocoproject.org/docs/Poco.Net.Socket.html

- WritableNotification – Do socketu je možné zapisovať dáta.
- ErrorNotification – Vznikla nejaká chyba na sockete.

SocketReactor pracuje v iteráciách, kde po krátky nastavený čas zisťuje zmeny na registrovaných socketoch a následne sériovo vyvolá obsluhu pre všetky sockety, na ktorých nastala nejaká udalosť.

Kapitola 4

Návrh a implementácia

Táto kapitola popisuje návrh a implementáciu serverovej časti označenej ako GWServer (Gateway Server), ktorý nahradí pôvodný ADA Server a novú komunikáciu s BeeeOn bránami.

Súčasná implementácia ADA Serveru a komunikácia s BeeeOn bránami sa ukázala ako nedostačujúca. Dôvodmi sú problémy s nasadením v domácnostiach, kvôli blokovaniu komunikácie rôznymi firewallmi sieťovej infraštruktúry. Ďalej vytváranie veľkého množstva nových spojení s bránami pre každú správu prichádzajúcu od brány, zatiaľ čo jedno spojenie s bránou je stále perzistentné, ale slúži len na zasielanie riadiacich príkazov bráne. Ďalším dôvodom je nevyhovujúca architektúra pôvodného serveru, s čím súvisí nemožnosť jednoduchého pridávania ďalších potrebných správ. Keďže na správy zasielané zo serveru v podobe riadiach príkazov brána nijako neodpovedá nie je zaručená spoľahlivosť, a či boli riadiace príkazy skutočne vykonané.

4.1 Požiadavky na nový spôsob komunikácie

Nová komunikácia medzi serverom a bránami, resp. pripojenými senzorickými a aktívnymi prvkami vyžaduje určité požiadavky, ktoré je pri návrhu potrebné zohľadniť. Keďže komunikácia prebieha cez Internet, je potrebné zabezpečiť prenášané dáta proti odpočúvaniu. Okrem toho je potrebná vzájomná autentifikácia servera a pripojených brán. Brána si potrebuje overiť, že komunikuje so správnym serverom a naopak na serveri sa musí overiť identita brány.

Z dôvodu jednoduchej integrácie do každej domácnosti je potrebné odstrániť problémy s firewallmi sieťovej infraštruktúry, resp. umožniť nasadenie systému BeeeOn aj v sieťach s obmedzeným prístupom na privilegované porty.

Je potrebné zaručiť spoľahlivosť doručenia dát, či už ide o zasielanie nameraných senzorických dát na server alebo doručenie riadiacich príkazov na bránu. Okrem toho je potrebné umožniť zistenie výsledkov príkazov a stav ich vykonávania. V pôvodnom systéme na riadiace príkazy brána nijako neodpovedala. Nebolo možné zistiť, či bol daný príkaz skutočne spracovaný a úspešne vykonaný. Okrem toho bol ignorovaný fakt, že vykonanie riadiacich príkazov (napr. nastavenie hodnoty aktívnemu prvku alebo odpárovanie zariadenia) sú typicky asynchrónne operácie a nemusia prebehnúť okamžite. Nová architektúra brány to však už umožňuje a preto je vhodné navrhnúť komunikáciu tak, aby bolo možné reportovať serveru stav vykonávania a dokončenie príkazu, a ďalej to zobraziť užívateľovi.

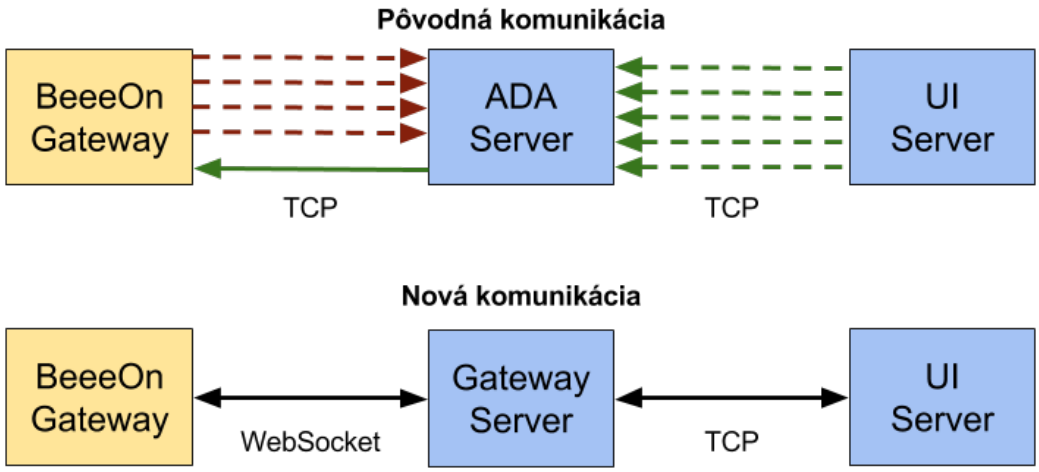
Výsledný server musí byť schopný obsluhovať potenciálne vysoký počet brán, s čím súvisí potrebná škálovateľnosť serveru. To znamená, že server musí byť konfigurovateľný podľa záťaže. Môže ísť o konfiguráciu počtu obslužných vlákien alebo spustenie serverovej časti Gateway Server na inom fyzickom stroji než zbytok serverovej aplikácie.

4.2 Návrh komunikácie

Pre komunikáciu medzi serverom a bránou som zvolil technológiu WebSocket zabezpečenú pomocou SSL/TLS, pretože umožňuje komunikovať i v sieťach s obmedzeným prístupom na privilegované porty a ponúka nízku réžiu prenášaných dát. Z dôvodu šetrenia systémových zdrojov je vytváranie nových spojení pre každú správu nahradené jedným perzistentným spojením.

Jedno perzistentné spojenie však prináša aj určité problémy. Keďže správy sú prenášané asynchrónne oboma smermi, nie je zaručené, že po zaslaní požiadavky príde odpoveď určená tejto požiadavke. Z tohto dôvodu som zaviedol unikátny identifikátor v každej správe, na základe ktorého je možné správne priradenie odpovede. Ďalším problémom je obsluha vysokého počtu perzistentných spojení a hlavne prijímanie dát z týchto spojení. Navrhnuté a implementované riešenie tohto problému popisuje kapitola 4.3.2.

Okrem komunikácie medzi bránami a serverom je potrebná aj zmena komunikácie medzi samotnými serverovými časťami. V pôvodnom systéme sa taktiež mrhalo systémovými prostriedkami a pre každý príkaz na nejakú bránu sa vytváralo nové spojenie. Navyše bola jedinou odpoveďou správa, či ADA Server príkaz odoslal alebo nie. Obrázok 4.1 znázorňuje porovnanie pôvodnej a navrhnutej komunikácie na najvyššej úrovni abstrakcie.



Obr. 4.1: Porovnanie pôvodnej a navrhnutej komunikácie

Komunikácia prebieha formou správ prenášaných vo formáte JSON. Pre jednoduché vytváranie a spracovanie správ, či už na serveri alebo bráne, som pre všetky potrebné správy vytvoril odpovedajúce triedy. Hlavnou triedou je trieda `GWMessage` a od nej dedia všetky ostatné triedy. Interná reprezentácia `GWMessage` je `Poco::JSON::Object`. Všetky parametre správy sa nastavujú priamo tomuto objektu a čítajú z neho. `GWMessage` je možné jednoducho pomocou jednej metódy previesť do textovej podoby na odoslanie a naopak vytvoriť z textovej reprezentácie.

4.2.1 Pripojenie brány k serveru

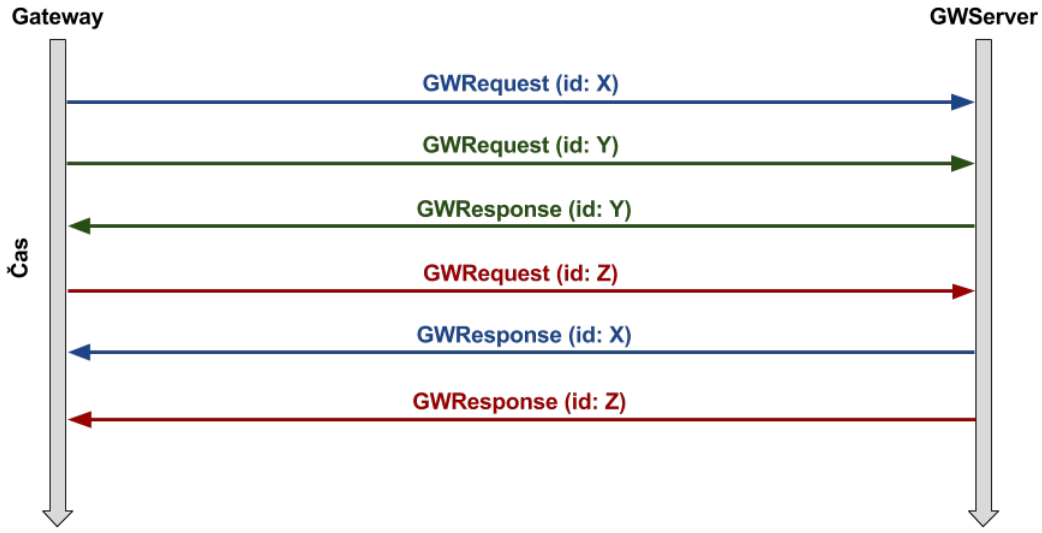
Zahájenie komunikácie prebieha pripojením brány k serveru. Brána sa typicky nachádza vo vzdialenej sieti v domácnosti bez verejnej IP adresy alebo za smerovačom s NATom. Bez konfigurácie smerovača alebo použitia VPN s ňou nie je možné jednoducho nadviazať spojenie zo strany serveru. Preto sa musí okamžite po zapnutí a strate spojenia pripojiť k serveru.

Ako prvé sa musí nadviazať WebSocket spojenie a následne zaslať registračná správa `GWGatewayRegister`. Obsahom správy je unikátny identifikátor brány, jej verzia a IP adresa. Odpoveďou od servera je správa `GWGatewayAccepted` označujúca, že server prijal spojenie a je pripravený s bránou komunikovať. V opačnom prípade je spojenie serverom uzavreté.

V prípade produkčného nasadenia je potrebné vytvorenie zabezpečeného spojenia pomocou SSL/TLS a overuje identitu brány z jej certifikátu.

4.2.2 Komunikácia iniciovaná z brány

Komunikácia iniciovaná z brány zobrazená na obrázku 4.2 prebieha formou požiadavka – odpoveď. Aby brána s ďalšou požiadavkou nemusela čakať kým dorazí odpoveď na predchádzajúcu požiadavku prebieha komunikácia asynchrónne. To znamená, že pre každú požiadavku sa na bráne vygeneruje unikátny identifikátor, ktorý sa pošle v požiadavke a následne ho server vloží do odpovedi. Okrem toho je potrebné, aby brána sledovala, či dorazí odpoveď do nejakého časového limitu. V prípade, že odpoveď nedorazí môže to znamenať, že bolo spojenie prerušené prípadne nekorektne ukončené. V tomto prípade je potrebné overiť, či prišla nejaká iná správa zo serveru v danom časovom limite, a ak nie pokúsiť sa vytvoriť nové spojenie.



Obr. 4.2: Komunikácia iniciovaná z brány

Všetky požiadavky a odpovede sú reprezentované triedami dediacimi od `GWRequest` resp. `GWResponse` a obsahujú unikátny identifikátor správy. `GWResponse` taktiež predstavuje generickú odpoveď na požiadavky, ktoré nevyžadujú žiadne dáta, len výsledok vykonania

požiadavky (úspech, neúspech). Nasledujúce kapitoly popisujú všetky požiadavky zasielané z brány spolu s odpoveďami od serveru.

Objavenie nového zariadenia

Po zapnutí párovacieho režimu na bráne, začne brána objavovať nové zariadenia. Tieto zariadenia je potrebné uložiť na serveri do databázy, aby sa mohli zobraziť užívateľovi. Ten si následne vyberie, ktoré zariadenia chce spárovať s bránou.

Požiadavku o objavení nového zariadenia reprezentuje trieda `GWNewDeviceRequest`. Obsahuje unikátny identifikátor zariadenia, názov a výrobcu pomocou čoho je možné identifikovať typ zariadenia, refresh time a zoznam typov modulov. Každý typ modulu je reprezentovaný typom (napr. teplota) a voliteľnými atribútmi (napr. vonkajší senzor).

Odpoveďou od serveru je generická odpoveď `GWResponse` informujúca o úspechu registrácie nového zariadenia.

Uloženie senzorických dát

Spárované zariadenia zasielajú namerané dáta zo svojich modulov za účelom ich uloženia v databáze. Frekvencia zasielania dát závisí od nastavenia refresh time.

Požiadavku na uloženie nameraných dát reprezentuje trieda `GWSSensorDataRequest`. Obsahuje unikátny identifikátor zariadenia, čas kedy boli dáta namerané a zoznam hodnôt. Hodnotu reprezentuje identifikátor modulu, nameraná hodnota a položka označujúca jej validitu.

Odpoveďou od serveru je generická odpoveď `GWResponse` informujúca o úspechu uloženia nameraných senzorických dát na serveri.

Získanie zoznamu spárovaných zariadení

Získanie spárovaných zariadení zo servera je potrebné pre správne fungovanie brány po jej reštarte. Brána konkrétne každý manažér zariadení si udržiava zoznam zariadení, ktoré sú s ním spárované a ktorými komunikuje. Po reštarte však tento zoznam stratí a musí si ho preto vyžiadať zo servera.

Požiadavka je reprezentovaná triedou `GWDeviceListRequest`. Obsahuje prefix identifikátora zariadení, ktoré vyžaduje. Pomocou prefixu sa vyfiltrujú práve zariadenia pre konkrétneho manažéra zariadení.

Odpoveďou od serveru je `GWDeviceListResponse` obsahujúca zoznam spárovaných zariadení a informáciu o úspechu. V prípade, že neexistujú žiadne spárované zariadenia s daným prefixom, výsledkom je prázdny zoznam a úspech.

Získanie poslednej nameranej hodnoty modulu zariadenia

Získanie poslednej nameranej hodnoty modulu zariadenia je potrebné po získaní zoznamu spárovaných zariadení. Pri aktívnych prvkoch je potreba im opäť nastaviť hodnotu, ktorú mali pôvodne nastavenú v prípade výpadku brány.

Požiadavka na získanie poslednej nameranej hodnoty modulu zariadenia je reprezentovaná triedou `GWLastValueRequest`. Obsahuje unikátny identifikátor zariadenia a identifikátor konkrétneho modulu.

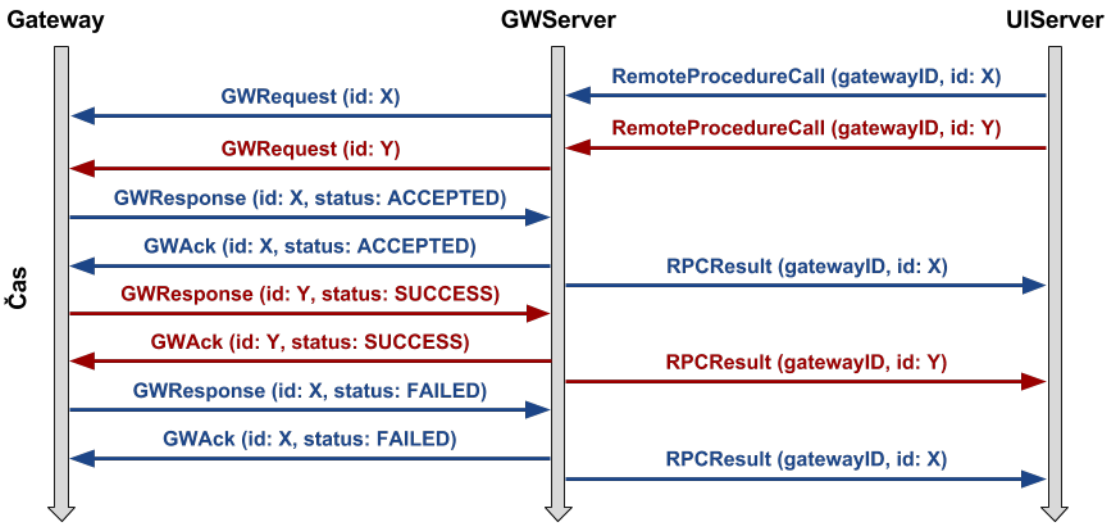
Odpoveďou od serveru je `GWLastValueResponse` obsahujúca informáciu o úspechu, nameranú hodnotu a či je hodnota validná. V prípade, že v databáze nie je uložená žiadna nameraná hodnota, odpoveďou je neúspech.

4.2.3 Komunikácia iniciovaná zo servera

Komunikácia iniciovaná zo servera zobrazená na obrázku 4.3 je komplikovanejšia. UIServer využíva na komunikáciu s bránami vzdialené volania cez GWServer. Vzdialené volania majú formu príkazov, ktoré sa majú vykonať na bráne, ale môžu mať asynchrónnu povahu. Každý príkaz môže priebežne oznamovať svoj stav spracovania. Aktuálne je možné oznámiť, že príkaz brána prijala na spracovanie a výsledok vykonávania príkazu. Do budúcna je možné odpovede rozšíriť o presnejšie informácie o stave vykonávania asynchrónnych príkazov. Každá požiadavka je identifikovaná unikátnym identifikátorom a bránou, na ktorú bola odoslaná.

GWServer po prijatí požiadavky na vzdialené volanie na bránu vytvorí `GWRequest` a odošle požiadavku na bránu. Ukladá si identifikátor požiadavky vygenerovaný na UIServeri, spolu s identifikátorom brány, na ktorú požiadavku odoslal. V prípade, že odpoveď nedorazí odošle UIServeru odpoveď, že brána neodpovedala. Ak žiadna iná správa neprišla od brány v danom časovom limite považuje ju za neaktívnu, zruší spojenie a brána musí vytvoriť nové. Po opätovnom pripojení je možné znova zaslať požiadavku bráne. V prípade doručenia odpovede od brány na požiadavku, ktorej vypršala platnosť, je odpoveď ignorovaná. Keď dorazí odpoveď od brány, GWServer jej potvrdí, že prijal odpoveď a prepošle odpoveď UIServeru.

Odpovede od brány zasielané na server sú potvrdzované, aby bola zaručená spoľahlivosť doručenia stavu vykonávania príkazov. Keďže brána zasiela niekoľko odpovedí na jednu požiadavku, je potrebné vzájomne priradiť odpovede a potvrdenia. Potvrdenie okrem identifikátora obsahuje aj označenie stavu z odpovede takže je dvojica správ jednoznačne identifikovaná. V prípade, že nedorazí potvrdenie od servera do časového limitu a taktiež ani žiadna iná správa je spojenie považované za neaktívne a brána sa pokúsi vytvoriť nové. Odpovede sú znova zasielané na server, ale v prípade viacerých odpovedí s rovnakým identifikátorom, je zasielaná len odpoveď s prioritnejším stavom.



Obr. 4.3: Komunikácia iniciovaná zo serveru

Odpovede od brány na príkazy zo serveru reprezentuje trieda `GWResponseWithAck`, z ktorej je možné jednoducho vytvoriť odpovedajúcu potvrdzovaciu správu `GWAck`.

Zistenie živosti brány

Na zistenie živosti brány slúži požiadavka `GWPingRequest`. Správa nemá žiadne ďalšie parametre a odpoveďou môže byť len úspech. Neúspech pri tomto type správy je práve žiadna odpoveď od brány do vypršania časového limitu.

Zapnutie párovacieho režimu na bráne

Zapnutie párovacieho režimu na bráne je potrebné pre objavenie nových zariadení. V tomto režime brána prijíma dáta aj zo zariadení, ktoré nie sú registrované v BeeeOn systéme a v prípade objavenia nových zariadení posiela požiadavky na registráciu nového zariadenia.

Požiadavku na zapnutie párovacieho režimu reprezentuje trieda `GWListenRequest`. Obsahuje dobu trvania párovacieho režimu. Po tejto dobe je párovací režim automaticky vypnutý.

Spárovanie zariadenia

Po objavení nových zariadení si užívateľ vyberie zariadenie, ktoré chce spárovať so svojou bránou. Zariadenie sa označí ako aktívne a na bránu sa pošle požiadavka na spárovanie daného zaradenia.

Požiadavku na spárovanie zariadenia reprezentuje trieda `GWDeviceAcceptRequest`. Obsahuje identifikátor zariadenia pre spárovanie.

Odpárovanie zariadenia

Odpárovanie zariadenia je potrebné v prípade, že sa užívateľ rozhodne, že dané zariadenie už ďalej nechce používať. Po odpárovaní zariadenia už z neho nebude možné prijímať dáta, až kým znova neprejde procesom párovania.

Požiadavku na odpárovanie zariadenia reprezentuje trieda `GWUnpairRequest`. Obsahuje identifikátor zariadenia pre odpárovanie.

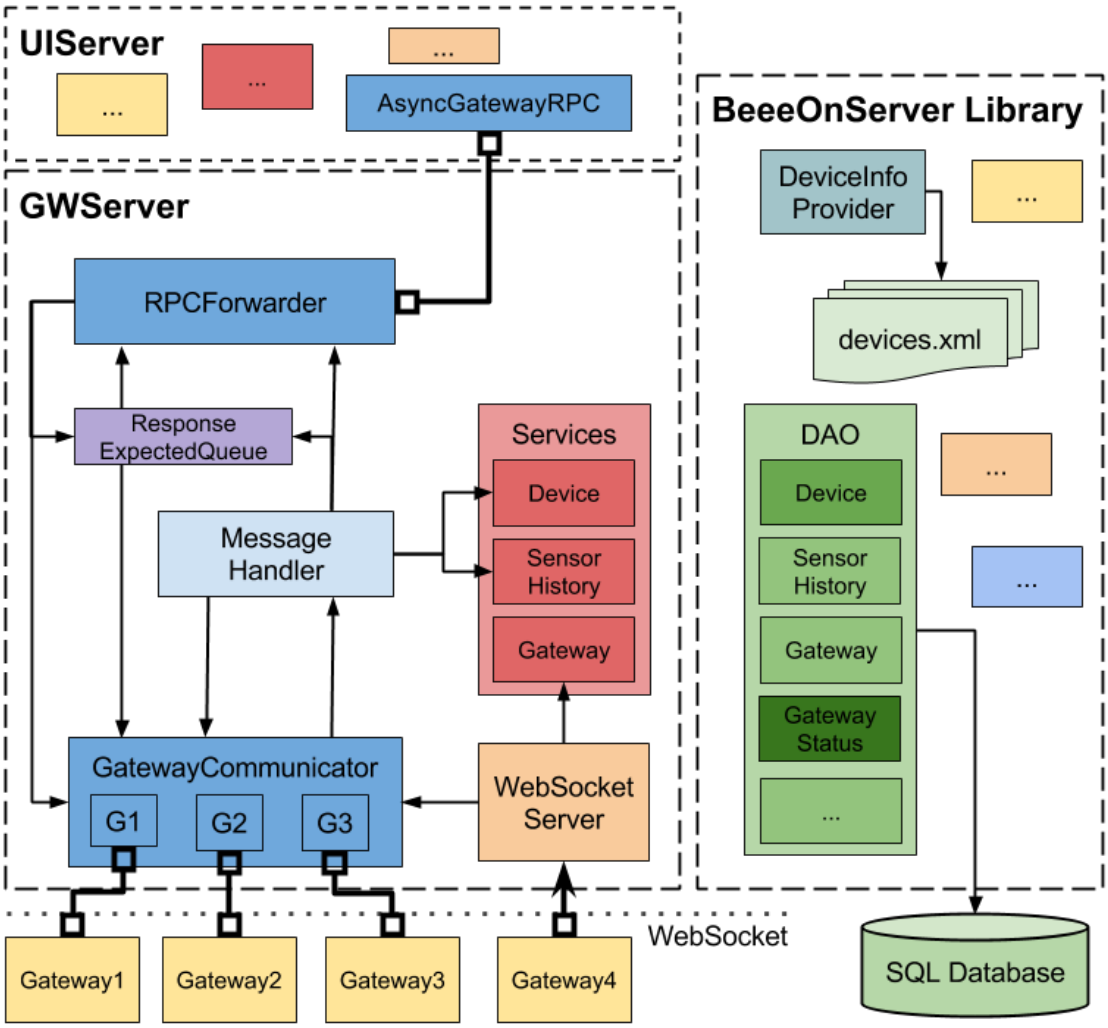
Nastavenie hodnoty aktívnemu prvku

Niektoré zariadenia obsahujú okrem senzorov aj aktívne prvky, ktorým je možné nastavovať hodnotu.

Požiadavka je reprezentovaná triedou `GWSetValueRequest`. Obsahuje identifikátor zariadenia, identifikátor modulu a hodnotu, ktorá sa má nastaviť.

4.3 Architektúra

GWServer je postavený na vrstvovej architektúre novo vyvíjaného BeeeOn serveru popísaného v kapitole 2.4.2. Využíva existujúci dátový model spolu s dátovou vrstvou. Obrázok 4.4 zobrazuje návrh architektúry GWServeru spolu s existujúcimi časťami BeeeOn serveru. Pre zjednodušenie sú zobrazené len potrebné časti dátovej vrstvy bez závislostí a prepojenie s UIServerom. Šípky označujú jednotlivé závislosti tried v mojom návrhu. Hrubými čiarami sú znázornené sieťové prepojenia. Jednotlivé triedy, ich prepojenia a konfigurácia sú popísané v nasledujúcich kapitolách.



Obr. 4.4: Návrh architektúry Gateway Serveru

4.3.1 WebSocketServer

Trieda `WebSocketServer` je zodpovedná za prijatie a vytvorenie WebSocket spojenia od brány, overenie jej identity, registráciu využitím príslušnej služby a následné pridanie spojenia do `GatewayCommunicatora`.

Využíva `Poco::Net::HTTPServer` framework. Po spustení vytvorí viac-vláknový HTTP server využívajúci ThreadPool. Server počúva na nastavenom porte a pomocou predanej `WebSocketRequestHandlerFactory` vytvorí príslušný `WebSocketRequestHandler` objekt pre každé nové spojenie. V ňom sa skonštruuje WebSocket a obslúži spojenie. Ak všetko prebehlo v poriadku, výsledkom je pridaná brána v `GatewayCommunicatore`.

Možnosti konfigurácie:

- **sslConfig** – SSL konfigurácia. V prípade, že nie je zadaná, server sa spustí bez šifrovania komunikácie pomocou SSL/TLS.

- **port** – Port, na ktorom server počúva.
- **backlog** – Maximálny počet čakajúcich spojení vo fronte na pripojenie.
- **maxMessageSize** – Maximálna veľkosť prichádzajúcej WebSocket správy.
- **minThreads** – Minimálny počet alokovaných vlákien ThreadPoolu WebSocket servera.
- **maxThreads** – Maximálny počet vlákien ThreadPoolu WebSocket servera.
- **threadIdleTime** – Špecifikuje dobu v sekundách, po ktorej je vlákno zničené v prípade, že je nečinné a počet bežiacich vlákien je väčší, ako minimálna kapacita Threadpoolu.

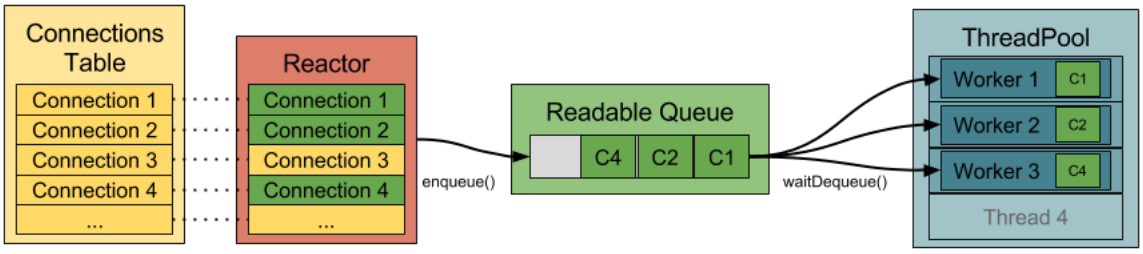
4.3.2 GatewayCommunicator

Trieda `GatewayCommunicator` je zodpovedná za komunikáciu s bránami po ich úspešnom pripojení a zaregistrovaní. V internej tabuľke si uchováva všetky spojenia s aktívnymi bránami. Spojenie s bránou predstavuje trieda `GatewayConnection` obsahujúca WebSocket tohto spojenia. `GatewayCommunicator` poskytuje ostatným častiam architektúry metódy na pridanie, odobranie a vyhľadanie spojenia s konkrétnou bránou. Pomocou tohto spojenia môžu ostatné časti zasielať správy bráne, ale za prijímanie správ a následnú obsluhu je zodpovedná táto trieda. Okrem tabuľky spojení, obsahuje `Poco::Net::SocketReactor` bežiaci v samostatnom vlákne.

Reactor na pozadí využíva systémové volanie *epoll* na zistenie prichádzajúcich dát alebo uzavretie spojenia. Pre túto detekciu je potrebné spojenie zaregistrovať do Reactora. Čítanie dát v implementovanom riešení však neprebieha v jeho vlákne, pretože môže byť blokujúce, kým sa neprečíta celá prichádzajúca správa. Funguje v iteráciách a v každej iterácií vyvolá sériovo obsluhu spojení, z ktorých je možné čítať dáta. Tieto spojenia sú postupne zaraďované do fronty a odregistrované z Reactora.

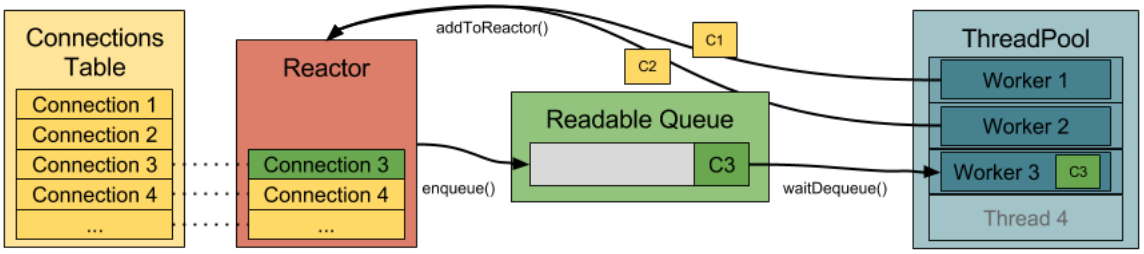
Poslednou časťou je konfigurovateľný ThreadPool, v ktorom sa podľa zátaže spúšťajú pracovné vlákna. Tieto pracovné vlákna si vyberajú spojenia z fronty, prečítajú správu a spojenie zaregistrujú naspäť do Reactora. Následne pomocou triedy `GWMessageHandler` obslúžia prichádzajúcu správu.

Nasledujúce obrázky 4.5, 4.6, 4.7 znázorňujú popísaný princíp obsluhy spojení.



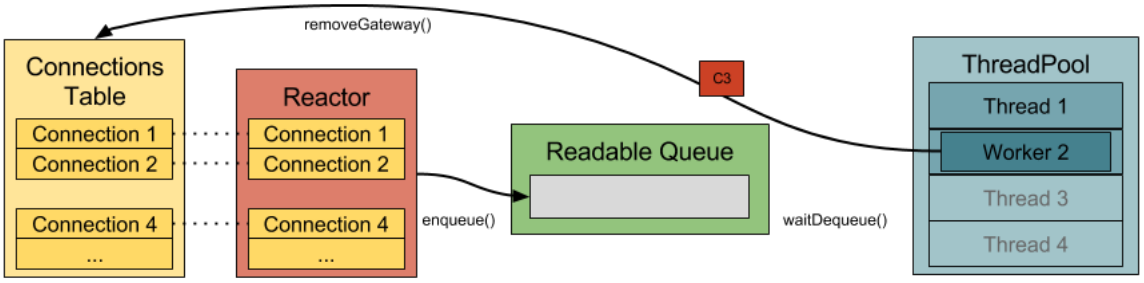
Obr. 4.5: Princíp fungovania triedy GatewayCommunicator

Obrázok 4.5 znázorňuje, že Reactor detekoval možnosť čítania dát zo spojení 1, 2 a 4. Tieto spojenia zaradil do fronty, spustili sa pracovné vlákna a začali ich obsluhovať.



Obr. 4.6: Princíp fungovania triedy GatewayCommunicator

Obrázok 4.6 znázorňuje stav, kedy pracovné vlákno 3 stihlo obsluhu oveľa skôr ako ostatné vlákna, preto začalo obsluhovať ďalšie pripravené spojenie vo fronte. Pracovné vlákna 1 a 2 dokončili obsluhu a vracajú spojenia do Reactora.



Obr. 4.7: Princíp fungovania triedy GatewayCommunicator

Posledný obrázok 4.7 znázorňuje, ako pracovné vlákno 2 zistilo, že sa spojenie uzavrelo a preto ho nevracia do Reactora, ale odstráni záznam z tabuľky spojení. Okrem toho znázorňuje stav kedy bol server spustený s nastavením minimálneho počtu pripravených vlákien v ThreadPoole triedy `GatewayCommunicator` na 2 vlákna. V tomto prípade sa vlákno nezničí, ale ostáva pripravené.

Možnosti konfigurácie:

- **maxMessageSize** – Maximálna veľkosť prichádzajúcej WebSocket správy.
- **receiveTimeout** – Doba, po ktorej bude vyhodená výnimka TimeoutException po začatí čítania WebSocket správy. Po zistení, že je možné čítať dáta zo socketu, je čítajúce vlákno blokované, až kým neprečíta celý rámec. V prípade, že by prijímanie rámca trvalo príliš dlho bude prerušené výnimkou.
- **sendTimeout** – Doba, po ktorej bude vyhodená výnimka TimeoutException v prípade bloknutia pri odosielaní správy.
- **minThreads** – Minimálny počet alokovaných vlákien ThreadPoolu.
- **maxThreads** – Maximálny počet vlákien ThreadPoolu.

- **threadIdleTime** – Špecifikuje dobu v sekundách, po ktorej je vlákno zničené v prípade, že je nečinné a počet bežiacich vlákien je väčší, ako minimálna kapacita ThreadPoolu.

4.3.3 GWMessageHandler

Trieda `GWMessageHandler` poskytuje rozhranie na obsluhu prijatej `GWMessage` správy. Implementáciu obsluhy obsahuje zdedená trieda `GWMessageHandlerImpl`. Prijatá správa môže byť požiadavka alebo odpoveď. Ak sa jedná o požiadavku, je zavolaná príslušná služba a odoslaná odpoveď bráne. V prípade, že sa jedná o odpoveď, je vybratá z fronty, bráne sa odošle potvrdenie a cez `RPCForwarder` je ďalej odpoveď preposlaná.

4.3.4 GWResponseExpectedQueue

Trieda `GWResponseExpectedQueue` umožňuje registráciu očakávaných odpovedí od brány a následnú reakciu na chýbajúcu odpoveď, po vypršaní časového limitu. V prípade, že odpoveď nedorazí, môže byť brána považovaná za neaktívnu. Je potrebné overiť, či v danom časovom limite prišla nejaká iná správa od brány. V prípade, že nie, je brána považovaná za neaktívnu a spojenie je ukončené. Zároveň je pomocou triedy `RPCForwader` zaslaný výsledok označujúci chýbajúcu odpoveď.

Trieda využíva `Poco::Util::Timer` a objekt `GenericPocoTimerTask`. Tento objekt predstavuje úlohu, ktorá sa vykoná v prípade chýbajúcej odpovede. Fronta si ho ukladá do internej tabuľky identifikovaný identifikátorom brány a požiadavky. Poskytuje metódy na pridanie očakávanej odpovede a zrušenie úlohy v prípade, že odpoveď dorazila.

Možnosti konfigurácie:

- **responseTimeout** – Doba, do ktorej by mala prísť odpoveď od brány na zaslanú požiadavku.

4.3.5 RPCForwarder

Trieda `RPCForwarder` poskytuje rozhranie na zaslanie odpovede od brány UIServeru. Implementáciu triedy obsahuje zdedená trieda `RPCForwarderImpl`. Jej úlohou je počúvať na serverom sockete na pripojenie UIServeru TCP spojením. Po pripojení transformuje prijaté požiadavky do podoby `GWRequest` a využitím triedy `GatewayCommunicator` ich zasiela na bránu. Zároveň zaregistruje očakávanú odpoveď do fronty `GWResponseExpectedQueue`.

Na prijímanie dát využíva `Poco::Net::SocketReactor`. Dáta sú čítané a spracovávané vo vlákne Reactora avšak kvôli možnosti zablokovania pri zasielaní požiadavky na bránu, sú využívané pracovné vlákna ThreadPoolu.

Odosielanie výsledkov príkazov UIServeru prebieha taktiež vo vlákne Reactora. Vyžíva sa k tomu fronta správ, ktoré sú postupne odosielané.

Možnosti konfigurácie:

- **port** – Port, na ktorom forwarder počúva.
- **maxMessageSize** – Maximálna veľkosť prichádzajúcej požiadavky.
- **minThreads** – Minimálny počet alokovaných vlákien ThreadPoolu.

- **maxThreads** – Maximálny počet vlákien ThreadPoolu.
- **threadIdleTime** – Špecifikuje dobu v sekundách, po ktorej je vlákno zničené v prípade, že je nečinné a počet bežiacich vlákien je väčší, ako minimálna kapacita ThreadPoolu.

4.3.6 AsyncGatewayRPC

Navrhol a implementoval som aj časť potrebnú na strane UIServera `AsyncGatewayRPC`. `AsyncGatewayRPC` poskytuje asynchrónne `GatewayRPC` pre UIServer. Podporuje asynchrónne zasielanie požiadaviek a prijímanie viacerých odpovedí na jednu požiadavku.

Po štarte sa pripojí na GWServer a umožní zasielať vzdialené volania na brány. V prípade výpadku spojenia sa automaticky znovu pripojí. Volajúci predá pri volaní ako parameter funkciu s parametrom `GatewayRPCResult`. Túto metódu si uloží do internej tabuľky identifikovanú vygenerovaným identifikátorom, ktorý sa posiela v požiadavke. Po doručení výsledku volania je daná funkcia vyvolaná.

Využíva `Poco::Util::Timer` a spolu s funkciou si ukladá aj príslušný `TimerTask`. V prípade, že výsledok nedorazí do časového limitu je zavolaná uložená funkcia s výsledkom TIMEOUT.

Pre prijímanie dát využíva `Poco::Socket::Reactor`, bežiaci v samostanom vlákne.

Možnosti konfigurácie:

- **host** – Adresa, na ktorú sa má rpc pripojiť.
- **port** – Port, na ktorý sa má rpc pripojiť.
- **maxMessageSize** – Maximálna veľkosť prichádzajúcej správy.
- **retryConnectTimeout** – Čas, po ktorom sa trieda znova pokúsi pripojiť v prípade, neúspešného pripojenia.
- **resultTimeout** – Čas, po ktorý musí prísť výsledok volania, inak je vrátený výsledok TIMEOUT.

4.4 Služby a rozšírenie dátovej vrstvy

Servisnú vrstvu serveru bolo potrebné rozšíriť o služby vyžadované pri pripojení brány k serveru, registráciu nových zariadení, zber senzorických dát a interné fungovanie brány. Potrebné služby sú popísané v nasledujúcich podkapitolách spolu s potrebnými rozšíreniami a úpravami dátovej vrstvy.

Registrácia brány

Registrácia brány je potrebná pri každom pripojení brány k serveru. V prípade prvého pripojenia sa s využitím dátovej vrstvy vytvára nová položka pre danú bránu, spolu s jej stavom. Stav reprezentuje čas posledného pripojenia, ip adresa a verzia brány. V prípade opakovanej registrácie sa ukladá len stav brány.

Pre uloženie stavu brány bolo potrebné rozšíriť dátovú vrstvu o `GatewayStatusDao`.

Registrácia nového zariadenia

Pri registrácií nového zariadenia je potrebné overenie, či už zariadenie nie je zaregistrované. V prípade, že by brána vyžadovala registráciu už zaregistrovaného zariadenia, zariadeniu sa len aktualizuje čas posledného videnia. V prípade novej registrácie je potrebné nastaviť čas prvého a posledného videnia na aktuálny čas a čas od kedy je zariadenie aktívne na null. Zároveň je potrebné skontrolovať validitu typu zariadenia a nakoniec využitím dátovej vrstvy uložiť zariadenie.

V pôvodnom systéme sa nové zariadenie pridávalo do databázy s prijatými dátami na základe typu v tabuľke *devices.xml*. Postupne, sa ale plánuje znížiť závislosť na tejto tabuľke a preto sa aktuálne zariadenie identifikuje menom produktu, výrobcom a zoznamom typov modulov. Súbor *devices.xml* som preto musel rozšíriť o tieto položky, na základe ktorých je možné zistiť typ zariadenia pre uloženie v databáze. Bolo potrebné upraviť aj DeviceInfoProvider, aby dokázal vyhľadať typ zariadenia na základe nového označenia.

Získanie zoznamu spárovaných zariadení

Pri získaní zoznamu spárovaných zariadení služba iba využije dátovú vrstvu na načítanie spárovaných zariadení s určitým prefixom.

Pre získanie zoznamu spárovaných zariadení bolo potrebné rozšíriť DeviceDao o metódu na získanie zariadení s daným prefixom.

Uloženie senzorických dát

Uloženie nameraných dát vyžaduje kontrolu existencie daného zariadenia asociovaného k bráne a následnú aktualizáciu času posledného videnia zariadenia. Ďalším krokom je kontrola, či identifikátory predaných modulov odpovedajú typu zariadenia. Ak všetko prebehlo v poriadku, dáta sú využitím dátovej vrstvy uložené.

Získanie poslednej nameranej hodnoty modulu zariadenia

Získanie poslednej nameranej hodnoty vyžaduje kontrolu existencie daného zariadenia asociovaného k bráne a overenie, že požadovaný modul odpovedá typu zariadenia. Následne sú dáta načítané z dátovej vrstvy.

Kapitola 5

Testovanie

Cieľom testovania bolo overiť funkčnosť serverovej aplikácie GWServer a jej jednotlivých častí, odhaliť chyby a následne ich opraviť. Testovanie prebiehalo priebežne počas celej doby implementácie. Pri vývoji som využíval jednorazové testy, zameriavajúce sa na konkrétnu implementovanú časť, spolu so sledovaním logovacích záznamov. Keďže server využíva viacero vlákien, odhalil som takto množstvo chýb spojených s paralelnou činnosťou a synchronizáciou.

Vytváranie JSON správ potrebných pre komunikáciu medzi bránou a serverom som testoval pomocou automatických jednotkových testov. Pre každú triedu reprezentujúcu konkrétny typ správy som vytvoril test na vytvorenie a parsovanie správy. Pri vytváraní správy som otestoval, či sa postupne vytváraná správa pomocou triednych metód na výstupe zhoduje s očakávanou JSON správou. Naopak pri parsovaní som otestoval zhodu položiek vstupnej JSON správy s položkami získanými pomocou triednych metód. Takto som odhalil hlavne chyby pri zložitejších správach obsahujúcich polia a štruktúry, na ktorých vytvorenie boli potrebné viaceré cykly.

5.1 Funkčné testy

Najdôležitejšou časťou testovania bolo otestovanie serverovej aplikácie ako celku. Pre testovanie som vytvoril sadu Python skriptov, ktoré simulovali požiadavky z brány aj z UIServera. Testy sú automatizované a vyžadujú spustenú inštanciu GWServera. Niektoré testy vyžadujú uložené údaje v databáze. Do databázy sa ukladajú testovacie dáta automaticky pri spustení servera v ladiacom režime. Jednotlivé testy sú automaticky vyhodnotené podľa očakávania. Pre bližšiu kontrolu, nie len očakávaného výsledku, ale aj priebehu operácií, je možné sledovať logovacie záznamy servera.

Test konektivity

Cieľom testu je overenie, že server počúva na zadaných portoch. Overuje sa pripojenie k strane komunikujúcej s bránami pomocou TCP aj vytvorenie WebSocketu. K druhej strane komunikujúcej s UIServerom sa vytvára TCP spojenie. Očakávaným výsledkom je úspešné vytvorenie a uzavretie spojenia.

5.1.1 Testovanie komunikácie iniciovanej z brány

Test registrácie brány

Cieľom testu je overenie registrácie brány na server po pripojení. Test sa skladá z 3 podtestov:

- V prvom teste sa zasiela registračná správa so všetkými validnými položkami. Očakávaným výsledkom je úspešná registrácia brány na serveri, prijatie potvrdzovacej správy a otvorené WebSocket spojenie.
- Druhý test sa pokúša registrovať bránu s jej nevalidným identifikátorom (vyžadovaných je 16 decimálnych číslic s určitými pravidlami). Očakávaným výsledkom je uzavretie spojenia.
- Tretí test overuje opakovanú registráciu brány. Očakávaným výsledkom sú obe úspešné registrácie, ale po opakovanej registrácií sa uzavrie pôvodné spojenie.

Test registrácie nového zariadenia

Cieľom testu je overenie úspešnosti registrácie nového zariadenia. Test sa skladá z 2 podtestov, ktoré vyžadujú najprv úspešnú registráciu pripojenej brány:

- Prvý test zasiela požiadavku na registráciu nového zariadenia s validnými položkami. Očakávaným výsledkom testu je odpoveď o úspešnej registrácií.
- Druhý test sa pokúša registrovať zariadenie, ktorého názov produktu a výrobcu server nepozná (nenachádza sa v tabuľke zariadení *devices.xml*). Test preto očakáva odpoveď o neúspešnej registrácií zariadenia.

Test uloženia senzorických dát

Cieľom testu je overenie požiadavky na uloženie senzorických dát na serveri. Pre úspešné uloženie musí byť zariadenie, z ktorého dáta pochádzajú registrované. Preto, je pred zaslaním senzorických dát vyžadovaná registrácia zariadenia a ihneď po pripojení samozrejme registrácia brány. Test obsahuje 3 podtesty:

- Prvý test zasiela validnú požiadavku na uloženie senzorických dát a očakáva odpoveď o úspešnom vykonaní operácie.
- V druhom teste sú zaslané dáta obsahujúce identifikátor modulu, ktorý daný typ zariadenia skutočne neobsahuje. Očakávaným výsledkom je teda odpoveď o neúspešnom uložení senzorických dát.
- Očakávaným výsledkom posledného testu je taktiež neúspech. Senzorické dáta v požiadavke sú asociované k zariadeniu, ktoré na serveri nie je registrované.

Test získania zoznamu spárovaných zariadení

Cieľom testu je úspešné získanie zoznamu spárovaných zariadení s konkrétnym prefixom. Testovacia databáza obsahuje niekoľko zariadení asociovaných k pripojenej bráne, pod ktorej identifikátorom sa v teste pripája k serveru. Práve 1 zariadenie má identifikátor obsahujúci tento prefix, preto je očakávaným výsledkom testu úspešná odpoveď s daným identifikátorom zariadenia.

Test získania poslednej nameranej hodnoty

Cieľom testu je úspešné získanie poslednej nameranej hodnoty na module určitého zariadenia. Podobne ako v predchádzajúcom teste je dotazované zariadenie uložené v testovacích dátach pri vytváraní databázy. Pred začiatkom testu je nutné najprv registrovať bránu. Očakávaným výsledkom testu je úspešná odpoveď s poslednou nameranou hodnotou skutočne uloženou v databáze.

5.1.2 Testovanie komunikácie iniciovanej zo servera

Testovanie komunikácie iniciovanej zo servera vyžaduje pripojenie klienta na RPC rozhranie GWServera, ktorý zasiela požiadavky na bránu s daným identifikátorom. Zároveň vyžaduje pripojenie klienta vystupujúceho ako brána, ktorý odpovedá podľa definovaného scenára.

Test zistenia živosti brány

Cieľom testu je overenie požiadavky na zistenie živosti brány. Test sa skladá z 3 podtestov:

- V prvom teste sa zasiela požiadavka na pripojenú bránu, ktorá odpovie. Očakávaným výsledkom je prijatie požiadavky na pripojenej bráne a po odpovedaní, prijatie úspešnej odpovede na serverovej strane.
- Druhý test zasiela požiadavku na bránu, ktorá nie je pripojená. Očakávaným výsledkom je odpoveď reflektujúca túto skutočnosť.
- Posledný test zasiela požiadavku na bránu, ktorá neodpovie. Očakávaným výsledkom testu je prijatie neúspešnej odpovede, po vypršaní časovača na očakávanú odpoveď.

Test zapnutia párovacieho režimu na bráne

Cieľom testu je overenie požiadavky na zapnutie párovacieho režimu. Brána odpovedá najpr potvrdením prijatia príkazu a následne zasiela odpoveď o úspešnom vykonaní príkazu. Očakávaným výsledkom testu je prijatie odpovede o akceptovaní požiadavky a následne o úspešnom vykonaní príkazu. Overuje sa taktiež, či brána prijala správnu požiadavku, ktorá jej bola zaslaná a taktiež prijatie potvrdzovacích správ po odoslaní odpovedí.

Test odpárovania zariadenia

Cieľom testu je overenie požiadavky na odpárovanie zariadenia. Test prebieha rovnako ako test zapnutia párovacieho režimu, ale namiesto úspechu je očakávaná neúspešná odpoveď.

Test spárovania zariadenia

Cieľom testu je overenie požiadavky na spárovanie zariadenia. Brána hneď zasiela odpoveď o neúspechu. Očakávaným výsledkom testu je prijatie úspešnej odpovede. Overuje sa taktiež, či brána prijala správnu požiadavku, ktorá jej bola zaslaná a taktiež prijatie potvrdzovacej správy po zaslaní odpovede.

Test nastavenia hodnoty aktívnemu prvku

Cieľom testu je overenie požiadavky na nastavenie hodnoty aktívnemu prvku. Test prebieha rovnako ako test spárovania zariadenia, ale namiesto neúspechu je očakávaná úspešná odpoveď.

5.2 Zhodnotenie testovania

Cieľom testovania bolo overiť funkčnosť serverovej aplikácie GWServer. Priebežným testovaním jednorazovými testami som odhalil množstvo chýb už pri vývoji, spojených hlavne s paralelizmom a chybami z nepozornosti. S využitím jednotkových testov som poriadne otestoval prácu s triedami GWMessage reprezentujúcimi všetky aktuálne potrebné správy medzi bránou a serverom. Následne som vytvorenou sadou funkčných testov otestoval aplikáciu z pohľadu rozhrania pre zbytok systému. Popísané funkčné testy odhalili niektoré chyby hlavne spojené so službami a prepojením s dátovou vrstvou. Pripisujem to hlavne tomu, že priebežné testovanie týchto častí bolo komplikované, pretože vyžadujú veľké množstvo závislostí. Chyby objavené pri testovaní som opravil, takže popísané scenáre aktuálne prechádzajú podľa očakávaní.

Kapitola 6

Záver

Cieľom bakalárskej práce bolo navrhnúť princíp komunikácie medzi serverom a vzdialenými senzormi a aktívnymi prvkami v systéme BeeeOn, ktorý zohľadňuje možnosť obsluhovať vysoký počet zariadení, spoľahlivosť doručovania dát, zabezpečenie prenášaných dát, množstvo rôznych meraných veličín, testovateľnosť a škálovateľnosť, a následne implementovať server využívajúci navrhnutý princíp komunikácie.

Prvá časť práce spočívala v zoznámení sa so systémom BeeeOn, s jeho jednotlivými časťami, ich prepojením, a s problematikou komunikácie so vzdialenými senzormi a aktívnymi prvkami v tomto systéme. Následne bolo potrebné naštudovať princípy zabezpečenej komunikácie pomocou technológie WebSocket, s možnosťou obsluhovať vysoký počet zariadení na serverovej strane. Naštudované poznatky sú zhrnuté v prvých dvoch kapitolách bakalárskej práce.

Na základe naštudovaných informácií som navrhol princíp komunikácie pomocou technológie WebSocket medzi serverom a bránami BeeeOn v domácnostiach, ktoré komunikujú so zariadeniami obsahujúcimi senzory a aktívne prvky. Následne som navrhol a implementoval server, využívajúci navrhnutý princíp komunikácie. Technológia WebSocket bola zvolená z dôvodu možnosti komunikovať na privilegovaných portoch, čím sa odstránili pôvodné problémy blokovania komunikácie rôznymi firewallmi v sieťovej infraštruktúre.

Schopnosť obsluhovať vysoký počet zariadení som dosiahol využítím systémového volania epoll v kombinácií s frontou spojení, z ktorých prichádzajú dáta a konfigurovateľným threadpoolom. Toto riešenie zaručuje tiež férovú obsluhu pripojených brán a pôvodné vytváranie nových spojení pre každú správu bolo možné nahradiť jedným perzistentným spojením. Z dôvodu obojsmernej asynchrónnej komunikácie, bolo potrebné zaviesť identifikáciu jednotlivých požiadaviek, aby k nim bolo možné priradiť správne odpovede. Pre zaručenie spoľahlivosti doručenia dát som implementoval systém potvrdzovania všetkých správ. Všetky potrebné správy medzi bránou a serverom som spracoval do objektovej podoby pre jednoduché vytváranie a parsovanie správ. Implementované triedy budú teda použité aj na bráne BeeeOn.

Keďže server BeeeOn je rozdelený na časť komunikujúcu s užívateľom (UIServer) a časť, ktorá bola úlohou tejto práce, komunikujúcu s bránami v domácnostiach (GWServer), navrhol a implementoval som nad rámec zadania aj komunikáciu medzi týmito časťami servera. Ďalej som nahradil vytváranie nových TCP spojení jedným perzistentným spojením s asynchrónnou komunikáciu. Oproti pôvodnej implementácií kedy na riadiace príkazy zasielané od užívateľa neprichádzali žiadne odpovede od brány, nová implementácia umožňuje viac úrovňové potvrdzovanie príkazov. Keďže niektoré príkazy (napr. nastavenie hodnoty

aktívnemu prvku) majú asynchrónnu povahu, je možné serveru reportovať stav vykonania danej úlohy.

Najčastejšie možné scenáre výsledného riešenia komunikácie, som otestoval pomocou automatických testov. Pre realizáciu testov bola vytvorená sada skriptov. S ich pomocou potom bola overená funkčnosť vytvorenej implementácie.

V projekte BeeeOn pokračujem ďalej a ďalšou mojou úlohou je implementácia druhej strany komunikácie na bráne BeeeOn.

Literatúra

[1] *BeeeOn - Main Page*. [Online; navštívené 28.07.2017].
URL https://www.beeeon.org

[2] *BeeeOn - Repositories*. [Online; navštívené 28.07.2017].
URL https://www.github.com/BeeeOn

[3] BeeeOn: *Architektúra systému*. [Online; navštívené 28.07.2017].
URL https://antdev.fit.vutbr.cz/redmine/projects/iot/wiki/architektura

[4] BeeeOn: *Brána*. [Online; navštívené 28.07.2017].
URL https://antdev.fit.vutbr.cz/redmine/projects/adapter/wiki

[5] BeeeOn: *Brána*. [Online; navštívené 28.07.2017].
URL https://beeeon.org/wiki/Gateway

[6] BeeeOn: *Server*. [Online; navštívené 28.07.2017].
URL https://antdev.fit.vutbr.cz/redmine/projects/server/wiki

[7] BeeeOn; Viktorin, J.: *Layered Server*. [Online; navštívené 28.07.2017].
URL https://antdev.fit.vutbr.cz/redmine/projects/server/wiki/Layered_Server#Dependency-injection

[8] Coplien, J.; Schmidt, D. C.: *Pattern Languages of Program Design*. Addison Wesley, 1995, ISBN 0-201-6073-4.

[9] Fette, I.; Melnikov, A.: *RFC 6455 - The WebSocket Protocol*. ietf.org, Dec 2011, [Online; navštívené 28.07.2017].
URL http://tools.ietf.org/html/rfc6455

[10] Fowler, M.: *Inversion of Control Containers and the Dependency Injection pattern*. [Online; navštívené 28.07.2017].
URL https://www.martinfowler.com/articles/injection.html

[11] GmbH., A. I. S. E.: *POCO C++ Libraries – A Guided Tour of The POCO C++ Libraries*. [Online; navštívené 28.07.2017].
URL https://pocoproject.org/docs/00100-GuidedTour.html

[12] GmbH., A. I. S. E.: *POCO C++ Libraries – Download*. [Online; navštívené 28.07.2017].
URL https://pocoproject.org/download/

[13] GmbH., A. I. S. E.: *POCO C++ Libraries – Features*. [Online; navštívené 28.07.2017].
URL http://pocoproject.org/features.html

[14] GmbH., A. I. S. E.: *POCO C++ Libraries – Multithreading*. [Online; navštívené 28.07.2017].
URL https://pocoproject.org/slides/130-Threads.pdf

[15] GmbH., A. I. S. E.: *POCO C++ Libraries – Network Programming*. [Online; navštívené 28.07.2017].
URL https://pocoproject.org/slides/200-Network.pdf

[16] GmbH., A. I. S. E.: *POCO C++ Libraries – Poco::Net::SocketReactor*. [Online; navštívené 28.07.2017].
URL https://pocoproject.org/docs/Poco.Net.SocketReactor.html

[17] GmbH., A. I. S. E.: *POCO C++ Libraries – Poco::Util::Timer*. [Online; navštívené 28.07.2017].
URL https://www.appinf.com/docs/poco/Poco.Util.Timer.html

[18] Olimex: *A10–OLinuXino–LIME*. [Online; navštívené 28.07.2017].
URL https://www.olimex.com/wiki/A10-OLinuXino-LIME

[19] Oppliger, R.: *SSL and TLS: Theory and Practice*. Artech House, 2009, ISBN 9781596934481.

[20] Richards, M.: *Software Architecture Patterns*. O'Reilly, 2015, ISBN 978-1-491-92424-2.