VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

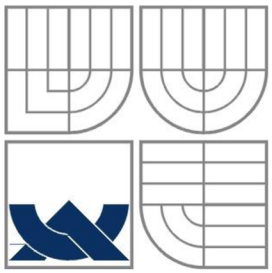
SERVER PRO INTELIGENTNÍ DOMÁCNOST

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

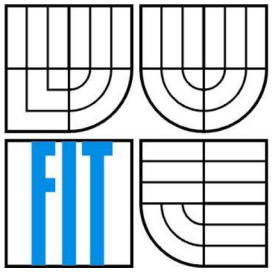
AUTOR PRÁCE
AUTHOR

MATÚŠ BLAHO

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SERVER PRO INTELIGENTNÍ DOMÁCNOST
SMARTHOME CLOUD

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MATÚŠ BLAHO

VEDOUCÍ PRÁCE
SUPERVISOR

ING PAVOL KORČEK

BRNO 2015

Abstrakt

Tato práce se zabývá návrhem a implementací aplikace serveru pro systém inteligentní domácnosti, která slouží na komunikaci se vstupním bodem domácnosti. Návrh a implementace se soustředí zejména na optimalizaci výkonu. Úlohou aplikace je sbírání a ukládání dat z domácnosti. Ukládání dat je realizováno pomoci databáze, s kterou aplikace komunikuje. Serverová aplikace je spouštěná jako služba běžící v pozadí z příkazové řádky. Implementace je realizovaná v jazyce C++.

Abstract

This thesis focuses on design and implementation of server for system of intelligent home. Main focus of design and implementation is performance optimization. Purpose of this application is to communicate with enter point of home. Task of this application is to gather and save data from homes. Storing of data is realized by database with witch the application communicates. Server application is started as service, running in background, from command line. Implementation is realized in language C++.

Klíčová slova

síťová aplikace, server, inteligentní domácnost, optimalizace výkonu

Keywords

network application, server, inteligent home, performance optimalization

Citace

Matúš Blaho: SERVER PRO INTELIGENTNÍ DOMÁCNOST, bakalářská práce, Brno, FIT VUT v Brně, 2015

SERVER PRO INTELIGENTNÍ DOMÁCNOST

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Pavla Korčeka.
Další informace mi poskytli Ing. Tomáš Novotný a Ing. Viktor Puš, PhD.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Matúš Blaho
17. 5. 2015

Poděkování

Chtěl bych poděkovat vedoucímu mé práce Ing. Pavlovi Korčekovi za odborné vedení, seznámení s tématem a pomoc při vypracovávaní bakalářské práce. Dále bych chtěl podekovat mému konzultantovi Ing. Viktorovi Pušovi, PhD za odbornou pomoc a celé výzkumné skupině, pod jejíž záštitou je tento projekt řešen.

Obsah

1 Úvod

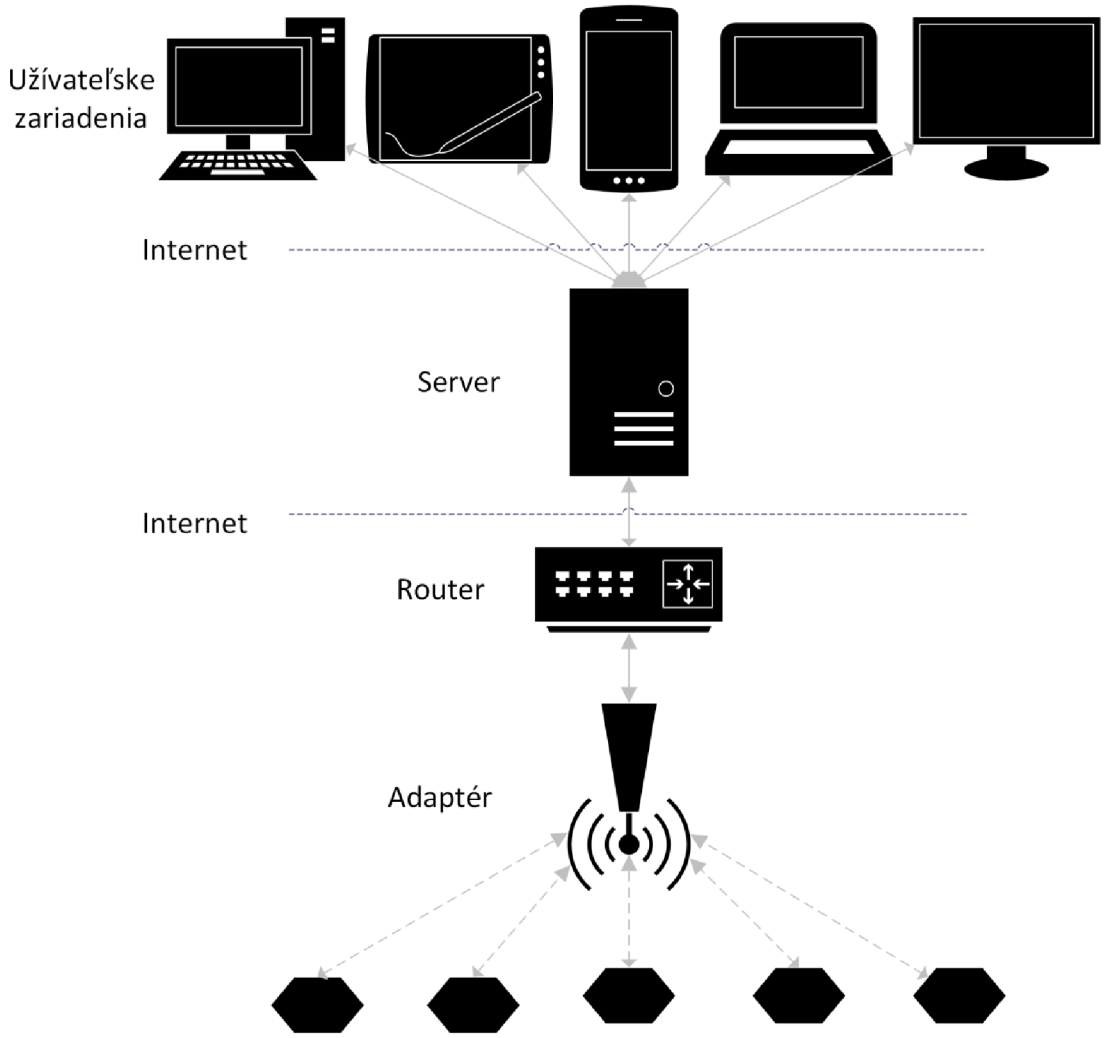
Internet začína s masovým nástupom postupne prenikať do nášho každodenného života. K internetu sa pomaly začínajú pripájať aj najzákladnejšie prvky nášho života. Preto je nutné prestať internet vnímať len ako zdroj informácií. Musíme internet začať vnímať aj ako prostriedok k ovládaniu vecí z reálneho sveta. Touto tematikou sa zaoberá „Internet of things“ skrátene IOT. Základnou predstavou tejto koncepcie je mať počítače schopné zbierať dáta z vecí reálneho sveta bez zásahu človeka [1]. Jednou z najdôležitejších vecí reálneho sveta je naša domácnosť. Využívame ju každý deň. Preto je veľmi zaujímavá vízia toho, že našu domácnosť budú sledovať inteligentné senzory, na základe ktorých budú aktívne prvky reagovať na vzniknuté situácie a udržiavať ju tak v nami požadovanom stave. Vďaka tomuto by sa postupom času mohla úloha človeka v starostlivosti o domácnosť obmedziť na konfiguráciu týchto zariadení a výmenu batérie v nich. Touto tematikou sa zaoberajú projekty inteligentných domácností. Samozrejme vo všetkých činnostiach potrebných pre bezproblémový chod domácnosti nie je možný nástup takejto technológie. Preto je nutné nasadzovanie systému rozdeliť podľa zložitosti jednotlivých činností od jednoduchých k zložitejším. Taktiež je treba zohľadniť technologickú realizovateľnosť. Návrhom a vytvorením jednoduchej inteligentnej domácnosti sa zaberá systém inteligentnej domácnosti vyvíjaný na FIT VUT. Náš projekt začína sledovaním fyzických podmienok v domácnosti ako je teplota vlhkosť a podobne. Základnými prvkami tohto systému sú senzory, ktoré snímajú aktuálny stav, ktorý si môže užívateľ zistiť a na základe neho nastaviť aktívne prvky. Projekt chce docieliť to, aby to mohol vykonávať aj bez fyzickej prítomnosti v domácnosti. Ako ideálnym prostriedkom sa stáva práve internet. Vzhľadom na to, že v dnešnej dobe je prístup na internet možný z telefónu alebo hodiniek, môže takýto vzdialený prístup k svojej domácnosti získať skoro každý. Ďalšou vecou, ktorú si musíme uvedomiť je, že inteligentná domácnosť sa nemusí nachádzať v inteligentnom dome. Pod inteligentným domov si môžeme predstaviť dom, ktorý bol navrhnutý a budovaný tak, že tieto technológie boli do neho zabudované od začiatku. Jedná sa teda najmä o kabeláž na prepojenie prvkov a ich napájanie. Prebudovanie obyčajného domu na inteligentný by bolo finančne aj časovo náročne. Stavebné úpravy eliminujú prvky inteligentnej domácnosti, ktoré sú bezdrôtové a ich napájanie je riešené napríklad zabudovaným akumulátorom. Ak by sa každý takýto prvok pripájal na internet osamostatnene, bolo by riešenie zložité a adaptácia na rôzne protokoly nemožná. Preto je nutné aby súčasťou domácnosti bol prvok ktorý bude preberať informácie zasielané prvkami a posielať ich ďalej cez internet. Tento prvok bude takisto tvoriť jedinečnú identifikáciu domácnosti. Tu nastávajú bezpečnostné problémy. Prvým z nich je, že rozsah bezdrôtovej siete nedokážeme obmedziť len medzi steny domu. Z toho dôvodu musí byť komunikácia medzi prvkami a výstupným bodom zabezpečená. Ďalším bezpečnostným rizikom je komunikácia prostredníctvom internetu. Nevyhnutnou súčasťou takéhoto systému aj je užívateľské rozhranie, cez ktoré bude užívateľ schopný získať informácie o svojej domácnosti, a nastavovať jej aktívne prvky. Prepojenie užívateľovej aplikácie na konkrétnu domácnosť môžeme riešiť centralizovane a decentralizovane. Decentralizované riešenie prináša komplikácie hlavne v tom, že užívateľ by musel mať pevnú IP adresu, ktorú by musel poznať. Ďalším problémom je to, že by musel mať úložisko dát a nastavení u seba doma alebo v danej aplikácii. Centralizované riešenie vyžaduje existenciu servera respektíve cloudu, ktorý bude združovať všetky tieto dáta a sprostredkovávať komunikáciu. Toto riešenie vyžaduje pevnú a známu IP adresu len pre tento server, keďže domácnosti aj užívatelia sa budú pripájať k nemu. Domácnosti sa budú jedinečne identifikovať pomocou výstupných prvkov

a užívatelia pomocou užívateľských účtov. Tento server tým pádom bude tvoriť pomyselnú chrbticu systému. Preto je nutné aby bol spoľahlivý. Takisto musí byť schopný obsluhovať veľký počet pripojení, s čo najnižším reakčným časom. Návrh, implementácia a riešenie problémov tejto časti systému je úlohou mojej bakalárskej práce a tento dokument slúži ako dokumentácia k nej. V kapitole 2 sa nachádza popis systému vyvíjaného na FIT VUT. Kapitola 3 obsahuje popis požiadaviek na spomínanú serverovú aplikáciu s návrhom ich riešenia. Implementačné detaily sú obsahom kapitoly 4. Nasleduje kapitola 5 popisujúca testovanie. Výsledky práce a ich zhodnotenie sa nachádza v kapitole 6.

2 Inteligentná domácnosť

2.1 Systém vyvíjaný na FIT VUT

Architektúra nami vyvíjaného systému sa skladá zo štyroch vrstiev (ako ilustruje Obrázok 2.1). Na najspodnejšej vrstve fungujú bezdrôtové koncové zariadenia, ktoré sa pripájajú na adaptér. Adaptér potom cez smerovač komunikuje so serverom. Na server sa z pomyslenej druhej strany pripájajú koncové zariadenia užívateľov.



Obrázok 2.1 – Schéma systému

2.1.1 Bezdrôtové koncové zariadenia

Tieto zariadenia sa skladajú z prvkov. Tie môžeme rozdeliť podľa určenia na 2 skupiny. Prvá skupina sú senzory, ktorých úlohou je snímať veličiny, druhou sú aktuátory, ktoré zasahujú do prostredia, a tak tieto veličiny ovplyvňujú. Zariadenia bezdrôtovo komunikujú s adaptérom, u ktorého sa registrujú pri spárovaní. Spárovanie môže byť vykonávané opakovane a jedná sa o užívateľsky jednoduchú operáciu (napríklad stlačenie tlačidla alebo zatrasenie prvkom). Bezdrôtová komunikácia je odolná voči odpočúvaniu (šifrovanie), rušeniu, a dokáže pokryť plochu obyčajného domu/bytu.

Každé z týchto zariadení má jedinečnú identifikáciu a vysiela o sebe základné údaje, ktoré využívajú vyššie vrstvy. Senzory sú pasívne prvky. Do prostredia domácnosti nijak nezasahujú len snímajú jeho stav. Sú napájané z batérie preto sú navrhnuté tak aby neboli spustené neustále ale v pravidelných intervaloch sa prebúdzali a odosielali hodnoty meraných veličín. Po ich odoslaní čakajú na správu s časom, po ktorého uplynutí sa majú opäť zobudiť a znovu zaslať hodnotu. V aktuálnom návrhu senzory zaznamenávajú jednu alebo viacero z nasledovných hodnôt : teplota, vlhkosť, tlak, zopnutie prepínača, intenzita svetla, intenzita hluku a emisie.

Aktuátory sú prvky, ktoré naopak ovplyvňujú stav domácnosti. Preto aktívne čakajú na správu o tom, že majú niečo vykonať. Z toho dôvodu, keď dôjde k zmene ich stavu z užívateľskej strany musí server zareagovať odoslaním informácie o tom, že sa stav zmenil. Bolo by nevyhovujúce keby aj tieto prvky fungovali na princípe prebúdzania. Ak by získanie ich nového stavu prebehlo až po prebudení, mohlo by to spôsobiť značné oneskorenie medzi rozhodnutím užívateľa zmeniť stav a jeho skutočným zmenením. Jedno koncové zariadenie v domácnosti môže obsahovať viacero aktívnych a pasívnych prvkov.

2.1.2 Adaptér a smerovač

Adaptér a smerovač slúžia ako brána do internetu pre bezdrôtové koncové prvky. Slúžia na posielanie dát medzi serverom a koncovými prvkami. Adaptér komunikuje len so zariadeniami, ktoré sa s ním spárovali. Realizácia pomocou jeho priamej implementácie do smerovača, by bola príliš zložitá, preto je implementovaný externe. Pomocou Ethernetu/WIFI sa v budúcnosti môžu k adaptéru pripájať aj iné komerčné adaptéry, s vlastnou podmnožinou koncových prvkov. Pre tieto zariadenia bude adaptér slúžiť ako prekladač protokolu na náš. Adaptér disponuje aj vlastným úložiskom dát, ktoré slúži v prípade výpadku internetovej konektivity. Vtedy sa do neho ukladajú údaje zasielané koncovými prvkami. V momente obnovenia pripojenia k serveru sú odoslané.

2.1.3 Server

Slúži ako medzivrstva medzi koncovými zariadeniami užívateľov a adaptérom. Jeho úlohou je zbierať dáta a ukladať ich. Dáta prijíma z obidvoch strán. Od užívateľov prijíma nastavenia aktuátorov a od adaptéra prijíma údaje od senzorov a stavy aktuátorov. Táto časť systému je pre všetky domácnosti spoločná, preto beží na externom serveri s verejnou IP adresou. Takéto riešenie prináša aj výhody, aj nevýhody. Výhodou je, že vďaka jeho verejnej IP adrese je pripájanie k nemu oveľa jednoduchšie. Takisto môže užívateľovi sprostredkovávať ďalšie služby, ako je napríklad predpoveď počasia, či na ňom môže fungovať samotná logika inteligentnej domácnosti. Hlavnou nevýhodou tohto riešenia je, že pri nedostupnosti pripojenia k internetu by inteligencia domácnosti prestala fungovať. Ďalšou z nevýhod je, že dáta budú uložené verejne na internete, preto je nutné ich zabezpečenie a autentizácia pri prístupe k nim. Požiadavky na tento server sú najmä nízka latencia, schopnosť zvládnuť vysoké množstvo pripojení, zabezpečenie dát a v podstate neustála prevádzkyschopnosť.

2.1.4 Koncové ovládacie zariadenia

Koncové zariadenia slúžia pre prístup užívateľov k ich inteligentnej domácnosti. A to k dátam z domácnosti, tak aj k možnosti nastavovať stavy jednotlivých aktuátorov. Jedná sa o aplikáciu s užívateľským rozhraním, ktorá môže bežať na klasickom PC, tablete, Smartphone, SmartTV alebo iných. Aplikácia je nezávislá na architektúre a operačnom systéme zariadenia.

3 Server

Serverovú časť projektu môžeme na abstraktnej úrovni rozdeliť na 2 hlavné úzko prepojené časti. Prvou z nich je dátová časť, ktorá slúži na uchovávanie dát o domácnostiach a užívateľoch. Druhou časťou je komunikačná časť, ktorá bude tieto dáta sprostredkovávať užívateľom a takisto zariadeniam v inteligentnej domácnosti. Komunikačná časť sa ďalej delí na časť komunikujúcu s domácnosťou a jej prvkami a časť komunikujúcu s aplikáciami užívateľov, zobrazujúcimi stav domácnosti. Nasledujúca časť textu bude pojednávať o možných návrhoch jednotlivých častí a ich finálnej podobe.

3.1 Komunikačná časť

Komunikačná časť je z dôvodu požiadavky na rýchlosť implementovaná v jazyku C/C++. Ako cieľová architektúra bola zvolená skupina systémov Unixového typu. Preto nebolo nutnosťou vytvoriť aplikáciu s prenositeľným kódom a mohli byť využité špecifické vlastnosti zvoleného systému. Jedná sa napríklad o schránky na sieťovú komunikáciu. Z toho vyplýva aj voľba TCP/IP protokolu na transportnej vrstve.

Ako už bolo spomenuté vyššie server komunikuje s dvoma stranami. Na jednej strane sa nachádzajú koncové zariadenia v domácnosti, komunikujúce cez adaptér, a na druhej zariadenia užívateľov. Obidve strany používajú rôzne protokoly a prenášajú odlišné dáta. Preto sa ukázalo ako najlepšie riešenie, pre každú stranu vytvoriť vlastnú serverovú aplikáciu. Eliminoval sa tým problém, ktorý by vznikal, keď by pri prijatí správy server nevedel určiť, o ktorý protokol sa jedná. Toto by vyžadovalo existenciu logiky, pomocou ktorej by server odlíšil o komunikáciu s akým zariadením sa jedná. Až potom by mohlo začať samotné spracovanie správy a jej obsluha. Táto operácia by priniesla spomalenie serverovej aplikácie, čo je nežiaduce. Serverové aplikácie búdu očakávať prichádzajúce spojenia na rôznych portoch, vďaka čomu každá zo strán vie, na ktorú z nich sa pripája. Prepojenie medzi týmito stranami servera bude zastrešuje jeho dátová časť a pre správy zasielané od užívateľov do domácnosti prechádzajú cez priamy kanál pomocou schránok .

Serverová časť komunikujúca s užívateľskými aplikáciami nie je predmetom tejto práce ale v určitej miere ovplyvňovala jej návrh. Jednalo sa najmä o návrh spoločnej dátovej časti.

3.1.1 Definícia požiadaviek na komunikačnú časť

V tejto časti bude popísaný priebeh komunikácie medzi severom a jednotlivými prvkami domácnosti a dáta prenášané medzi nimi. Z toho následne vyplývajú požiadavky na komunikačnú časť servera. Komunikácia s koncovými prvkami závisí na ich vlastnostiach popísaných v kapitole 2.1.1.

Potreba oboznamovania aktívnych prvokov okamžite po ich zmene stavu viedla k novému druhu požiadaviek, ktoré musí aplikácia spracovávať. Tieto akcie sú zaznamenávané aplikáciou zastrešujúcou komunikáciu s užívateľskými zariadeniami. Tá musí informáciu predať nášmu serveru. Komunikácia musí prebiehať medzi aplikáciami priamo. Keby prebiehala len pomocou dátovej časti odozva pre užívateľské akcie by narástla. Prinieslo by to takisto nutnosť vytvorenia určitej logiky v dátovej časti, ktorá by sledovala zmeny dát a stavov. Preto sa aplikácia rozdelila na dve logické

časti. Prvá očakáva pripojenia zo strany domácnosti. Druhá očakáva pripojenie zo strany servera spracovávajúceho správy od užívateľských zariadení.

Koncové prvky a server nekomunikujú nikdy priamo. Ich komunikáciu sprostredkováva adaptér. Na základe toho vzniká medzi adaptérom a serverom systém správ. Ďalšími správami sú správy na obsluhu adaptéra. Jedná sa o správy vyvolané užívateľom. Ide o akciu nazývanú párovanie a mazanie. Pri akcií párovanie sa adaptér uvedie do stavu pri ktorom sa do jeho siete môžu pripojiť nové koncové prvky. Pri mazaní adaptér dané zariadenie zo svojej siete odstráni.

Systém je navrhnutý tak, že jediný jeho prvok s verejnou IP adresou je server. Adaptér sa preto môže nachádzať za smerovačom s prekladom sieťový adries (z angličtiny NAT network address translation). Preklad sieťových adries je funkcia, ktorá umožňuje prekladanie adries. Znamená to, že adresy z lokálnej siete sa preložia na jedinečnú adresu, ktorá slúži pre vstup do inej siete (napr. Internetu), prekladanú adresu si uloží do tabuľky pod náhodným portom, pri odpovedi si v tabuľke vyhľadá port a pošle pakety na IP adresu priradenú k danému portu. NAT je vlastne jednoduchým proxy serverom[2]. Náhodnosť portov spôsobuje, že nebude možné s adaptérom nadviazať spojenie iniciované zo strany servera. Vyriešiť je to možné mapovaním adries na smerovači. Vtedy sa konkrétnej vnútornej adrese pridelí konkrétny vonkajší port, z ktorého sú všetky dáta posielané na ňu. Toto riešenie však nespĺňa požiadavku na jednoduchú inštaláciu systému. Vyžaduje totiž od užívateľov znalosť nastavení smerovača a zásah do nich. Na základe toho je nutné nájsť vhodné riešenie na softwarovej úrovni medzi adaptérom a severom.

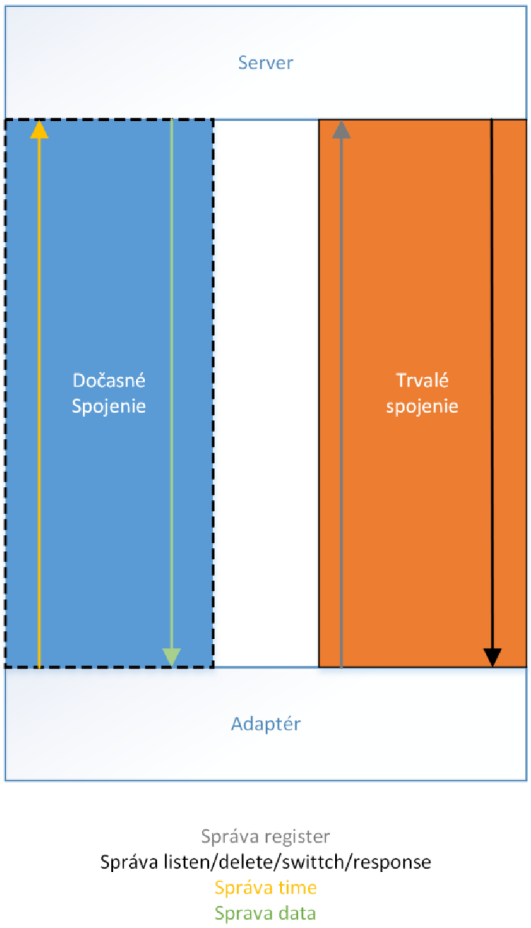
3.1.2 Model komunikácie a jej zabezpečenie

Z vyššie uvedených dôvodov bolo nutné navrhnúť model komunikácie, ktorý by umožňoval zasielať správy zo servera na adaptér. Jedným z riešení bolo softvérovo vytvoriť sieť, v ktorej by sa nachádzal adaptér aj server. To by umožnilo iniciovať spojenia zo serveru priamo na adaptér. Takúto možnosť poskytuje použitie virtuálnej privátnej siete (VPN). Virtuálna privátna sieť je v informatike prostriedok k prepojeniu niekoľkých počítačov prostredníctvom (verejnej) nedôveryhodnej počítačovej siete. Je možné tak jednoducho dosiahnuť stav, kedy spojené počítače budú medzi sebou komunikovať, akoby boli prepojené v rámci jediné uzavretej privátnej (a teda dôveryhodnej) siete. Pri nadviazaní spojenia je totožnosť oboch strán overovaná pomocou certifikátov, dôjde k autentizácii, celá komunikácia je šifrovaná, a preto môžeme také prepojenie považovať za bezpečné [3]. V tomto riešení je na serveri spustený VPN server. Každý adaptér vystupuje v úlohe VPN klienta a pri spustení sa pripojí k VPN serveru. Od tohto momentu sa nachádzajú server aj adaptér v jednej virtuálnej sieti, a tak aj server môže nadväzovať spojenie s adaptérom. Ďalšou z výhod tohto riešenia je, že komunikácia je šifrovaná priamo v implementácii VPN, a preto nie je potrebné ju implementovať v aplikáciách servera a adaptéra. Toto riešenie sa však ukázalo pre náš systém ako nevyhovujúce. Pri manipulácií s aplikáciou adaptéra a autentifikačnými certifikátmi by mohol užívateľ jednoducho zasielať do systému podvrhnuté správy.

Druhou z možností bolo využitie permanentného spojenia iniciovaného adaptérom. Pri tomto návrhu sa adaptér pripojí k serveru. Cez toto spojenie posiela správy a spojenie sa nikdy neuzatvorí ani jednou zo strán. V prípade príchodu užívateľskej požiadavky, ju server pošle cez toto spojenie na adaptér. Detekcia výpadku spojenia je vykonávaná adaptérom. V takej situácií sa adaptér opäť pripojí na sever a spojenie tak obnoví. Aj toto riešenie prináša určité nevýhody. Vyžaduje totiž od serverovej aplikácie, aby bola pripravená prijímať správy z mnohých spojení súčasne. To umožňujú v systémoch rodiny UNIX systémové funkcie poll respektíve select. Avšak tieto funkcie neumožňujú dynamické editovanie spojení, z ktorých majú očakávať správy. Táto možnosť je ale nevyhnutnosťou. Do

systému sa budú pripájať nové domácnosti a pri prerušení spojenia sa budú znovu pripájať aj domácnosti, ktoré sa už v systéme nachádzajú. Pri prerušení spojenia je takisto potrebné tieto spojenia ukončiť na strane servera. Tieto možnosti prináša systémová funkcia epoll. Ďalším problémom tohto riešenia je, že pri zasielaní správ z jedného adaptéra môže nastať situácia, keď dva alebo viac koncových prvkov pošle dáta. Pri súbežnej obsluhe požiadaviek koncových zariadení sa adaptér pokúsi poslať viac správ naraz a môže dôjsť k pomiešaniu ich obsahu.

Na základe vyššie uvedených dôvodov bol tento model komunikácie upravený do podoby, ktorú znázorňuje Obrázok 3.1. Princíp spočíva v tom, že medzi serverom a adaptérom existujú dva druhy spojení. Prvým je spojenie permanentné. Toto spojenie slúži na posielanie správ zo servera na adaptér. Spojenie je iniciované adaptérom. Adaptér cez neho pošle registračnú správu. Následne na tomto spojení očakáva správy od servera. Pri výpadku spojenia sa adaptér opäť pripojí k serveru a spojenie tak obnoví. Druhým druhom sú spojenie dočasné. Cez toto spojenie zasiela adaptér dáta koncových prvkov. Server mu ako odpoveď pošle čas do ďalšieho zobudenia koncového zariadenia. Spojenie sa následne ukončí. Pri ďalšej potrebe poslať dáta sa nadviaže nové spojenie.



Obrázok 3.1Návrh komunikačného modelu

Implementácia zabezpečenia musí byť vykonaná v aplikáciách adaptéra a servera. Ako zabezpečenie bolo zvolené SSL (*Secure Sockets Layer*). SSL je v počítačových sieťach protokol, ktorý sa stará o autentizáciu servera, klienta a kryptovanú komunikáciu medzi servermi a klientmi.[4]

Ustanovenie SSL spojenia funguje na princípe asymetrickej šifry. Každá z komunikujúcich strán má dvojicu šifrovacích kľúčov - verejný a súkromný. Verejný kľúč je nutné zverejniť a zaistiť jeho správne predanie všetkým, ktorí ho budú chcieť použiť. Pokiaľ pomocou tohto kľúča hocikto

zašifruje správu, je zaistené, že ju bude môcť rozšifrovať len majiteľ použitého verejného kľúča odpovedajúcim súkromným kľúčom [5]. Použitie SSL umožňuje aby súčasťou verejného kľúča adaptéra bola aj jeho unikátna identifikácia v rámci systému. Server tak dokáže overiť identitu adaptéru voči identifikátoru v správe, a preto ani pri manipulácií s aplikáciou adaptéra nemôže adaptér posielať podvrhnuté dáta predstierajúc, že je iný adaptér.

3.1.3 Protokoly používané pre komunikáciu servera

V tejto kapitole bude popísaný protokol a jeho správy, navrhnuté pre komunikáciu servera s adaptérom a na druhej strane so serverom obsluhujúcim užívateľské požiadavky. Protokol je postavený na jazyku XML[6].

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <adapter_server protocol_version="0.1" state="data"
    adapter_id="6499655" fw_version="0" time="1403173390">
3.      <device id="123456789ab">
4.          <battery value="40"/>
5.          <rssi value="80"/>
6.          <values count="3">
7.              <value type="0x0A" offset="0x00">25.5</value>
8.              <value type="0x0A" offset="0x01">26.0</value>
9.              <value type="0x01" offset="0x00">67.0</value>
10.   </values>
11. </device>
12. </adapter_server>
```

Zdrojový kód 3.1: Šablóna XML správy zasielanej adaptérom

V ukážke zdrojový kód 1 sa nachádza šablóna XML správy od adaptéru. Základom správy je element `adapter_server`. Atribút `protocol_version` vyjadruje verziu komunikačného protokolu používanú adaptérom. Atribút `adapter_id` nesie hodnotu jedinečného identifikátoru adaptéru v systéme inteligentnej domácnosti. Jeho dátový typ je kladné číslo v rozsahu 64 bitov zapísanom v hexadecimálnom formáte. Atribút `fw_version` obsahuje informáciu o verzií firmvéru adaptéra. Čas odoslania správy je vyjadrený pomocou atribútu `time` vo formáte časového razítka systému UNIX. Atribút state nesie informáciu o druhu správy. Môže nadobúdať hodnotu `data`, pre správu nesúcu informáciu o dátach z domácnosti, alebo register, pre registračnú správu posielanú pri pripojení adaptéra.

Registračná správa neobsahuje element device. Element device sa v správe nachádza len pri posielaní dát z domácnosti. Jeho atribút `id` nesie jednoznačný identifikátor zariadenia. Jeho rozsah je kladné číslo v rozsahu 32 bitov. Element `battery` a jeho atribút `value` zaznamenávajú hodnotu batérie zariadenia v percentách. Element `rssi` s atribútom `value` informujú o hodnote kvality signálu takisto v percentách. Atribút `count` elementu `values` vyjadruje počet prenášaných hodnôt. Od tohto počtu sa odvíja aj počet podelementov `value`. Element `value` slúži na prenos hodnoty. Atribút type vyjadruje typ a na ňom závisí aj hodnota samotného elementu. Tabuľka 3.1 zobrazuje číselné kódy používané pre jednotlivé zariadenia. V treťom stĺpci sa nachádza dátový typ hodnoty. Vzhľadom na to, že jeden koncový prvok môže zaznamenávať viac hodnôt toho istého typu obsahuje element `value` ešte atribút `offset`. Kombinácia type a `offset` vytvárajú jedinečnú identifikáciu hodnoty v koncov prvku.

Hodnota typu	Veličina	Dátový typ hodnoty a jeho veľkosť
0x01	Senzor vlhkosti	Celočíselný
0x02	Senzor tlaku	Celočíselný
0x03	Senzor Otvorenia/Zatvorenia	Bit 0/1
0x04	Senzor Zapnutia/Vypnutia	Bit 0/1
0x05	Senzor intenzity svetla	S plávajúcou desatinnou čiarkou
0x06	Senzor intenzity hluku	S plávajúcou desatinnou čiarkou
0x07	Senzor CO2 v ovzduší	Celočíselný
0x08	Senzor snímajúci pozíciu	Bitové pole
0x0A	Senzor teploty	S plávajúcou desatinnou čiarkou
0x0B	Senzor statusu bojlera	Celočíselný
0xA0	Aktuátor zapnutie/vypnutie	Bit 0/1
0xA1	Aktuátor zapnutie	Bit 0/1
0xA2	Aktuátor prepnutia stavu	Bit 0/1
0xA3	Aktuátor nastavujúci rozsah	Celočíselný
0xA4	Aktuátor RGB	Celočíselný
0xA5	Hodnota teploty bojlera	S plávajúcou desatinnou čiarkou
0xA6	Typ operácie bojlera	Celočíselný
0xA7	Mód operácie bojlera	Celočíselný

Tabuľka 3.1 - Popis významu dvojíc typ a hodnota

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <server_adapter protocol_version="0.1" state="set" id="123456789ab"
   time="0" type="0x04" offset="0">
3.     <value>1</value>
4. </server_adapter>
```

Zdrojový kód 3.2: Šablóna správy posielanej zo servera na adaptér

Správy posielané zo servera na adaptér sú stručnejšie. Ich šablóna sa nachádza v ukážke Zdrojový kód 3.2. Základom správy je prvok `server_adapter`. Ten obsahuje atribút `protocol_version` značiaci verziu komunikačného protokolu. Atribút state nesie informáciu o druhu správy. Tieto dva atribúty sú základné.

Druhy správ podľa hodnoty atribútu state:

- `Listen` – správa oznamujúca adaptéru, že sa má prepnúť do režimu párovania. Táto správa je vyvolaná užívateľom.
- `Clean` – správa oznamujúca adaptéru, že má odstrániť zo svojej siete koncový prvok. Obsahuje navyše atribút `id` informujúci adaptér, o ktorý koncový prvok sa jedná. Táto správa je vyvolaná užívateľom.
- `Register` – správa je odpoveďou na registračnú správu adaptéra. Obsahuje atribút `response`, ktorý ma pri úspechu registrácie hodnotu `true`. V prípade neúspechu hodnotu `false`.
- `Time` – správa je odpoveďou na správu od adaptéra nesúcu dáta. Obsahuje atribút `time`, ktorý nesie informáciu o dĺžke časového úseku do ďalšieho zobudenia koncového prvku v sekundách.
- `Switch` – správa slúžiaca na prepnutie aktuátora. Obsahuje atribút `id` informujúci adaptér, na ktorom koncovom prvku má aktuátor zmeniť svoju hodnotu. Obsahuje

takisto atribút `type`, jeho hodnoty a ich význam sa nachádzajú v tabuľke 3.1. Posledným atribútom je `offset`. Súčasťou správy je ešte prvok `value` vložený v základom prvku správy. Tento prvok nesie informáciu o hodnote, na ktorú sa ma aktuátor nastaviť.

```
1. <request type="switch">
2.     <sensor id="1111" type="0x00" onAdapter=12345">
3.         <value>0</value>
4.     </sensor>
5. </request>
```

Zdrojový kód 3.3 : Šablóna správy prijímanej zo servera obsluhujúceho užívateľov

Správy posielané zo servera obsluhujúceho užívateľské aplikácie majú ako základ prvok `request`. Jeho jediným atribútom je type. Šablónu tejto správy ilustruje Zdrojový kód 3.3.

Druhy správ podľa atribútu type:

- `Listen` – Správa oznamujúca serveru aby poslal na adaptér požiadavku na prepnutie do režimu párovania. Základný element obsahuje jediný podelement `adapter` s atribútom id. Atribút id nesie jedinečný identifikátor adaptéru, na ktorý sa ma poslať požiadavka.
- `Delete` - Správa oznamujúca serveru aby poslal na adaptér požiadavku na odstránenie koncového prvku. Základný element obsahuje jediný podelement `senzor`, s atribútmi `id` a `onAdapter`. Atribút `id` nesie identifikáciu zariadenia, ktoré sa ma odstrániť. Atribút `onAdapter` nesie identifikáciu adaptéru, ktorému bude požiadavka zaslaná.
- `Swtich` - Správa oznamujúca serveru aby poslal na adaptér požiadavku nastavenia aktuátoru. Základný element obsahuje jediný podelement senzor, s atribútmi `id`, type a `onAdapter`. Atribút `id` nesie identifikáciu zariadenia, na ktorom sa aktuátor nachádza. Atribút type obsahuje informáciu o aktuátore vramci tohto zariadenia. Je v ňom zakódovaná informácia o `type` a takzvanom posunutí, ktoré v správe na prepnutie aktuátoru posielanej do domácnosti vystupujú ako atribúty type a `offset`. Atribút `onAdapter` identifikuje adaptér, ktorému bude požiadavka zaslaná.

3.1.4 Voľba spôsobu obsluhy požiadaviek klienta

Následne bolo nutné zvoliť spôsob obsluhy klientskych požiadaviek serverom. Pri vytváraní serveru TCP rozlišujeme dva typy serverov: iteratívny server TCP, ktorý spracováva požiadavky postupne, a konkurentný server TCP, ktorý ich spracováva súbežne [7]. Následne si popíšeme a priblížime ich princíp fungovania.

Algoritmus práce iteratívneho serveru:

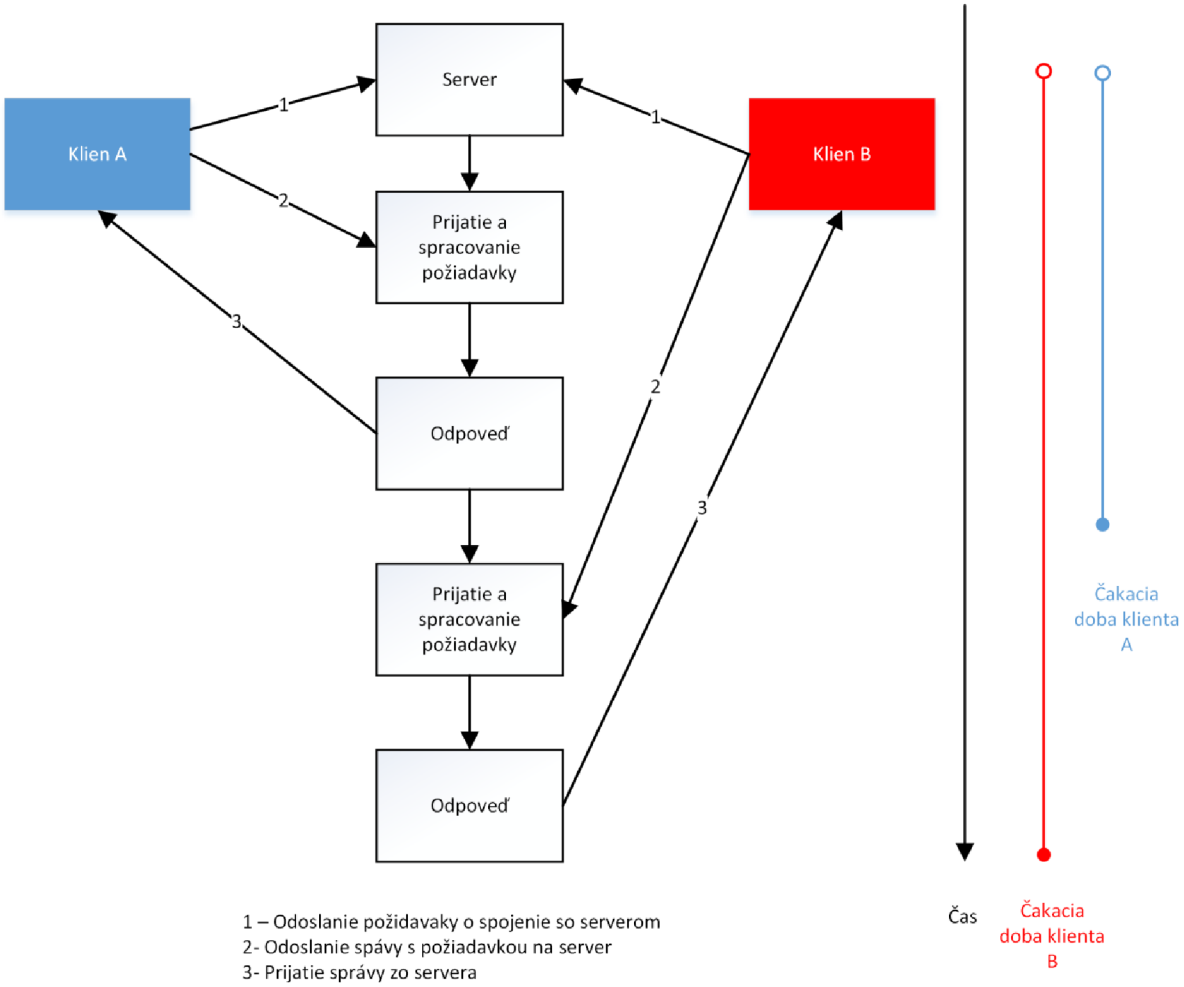
1. Otvor schránku a spáruj ju s požadovaným portom.
2. Prepni schránku do pasívneho režimu.
3. Prijmi požiadavku a vytvor novú schránku na komunikáciu s klientom.
4. Prijmi správu od klienta a spracuj ju.
5. Odpovedz klientovi.
6. Uzatvor komunikačnú schránku a vráť sa na krok 3.

Algoritmus práce konkurentného serveru:

1. Otvor schránku a spáruj ju s požadovaným portom.
2. Prepni schránku do pasívneho režimu.

3. Prijmi požiadavku a vytvor novú schránku na komunikáciu s klientom.
4. Vytvor nový proces.
5. V hlavnom procese uzatvor komunikačnú schránku a vráť sa na krok 3.
6. Vo vedľajšom procese prijmi správu od klienta a spracuj ju.
7. Vo vedľajšom procese odpovedz klientovi.
8. Vo vedľajšom procese uzatvor komunikačnú schránku a ukonči proces.

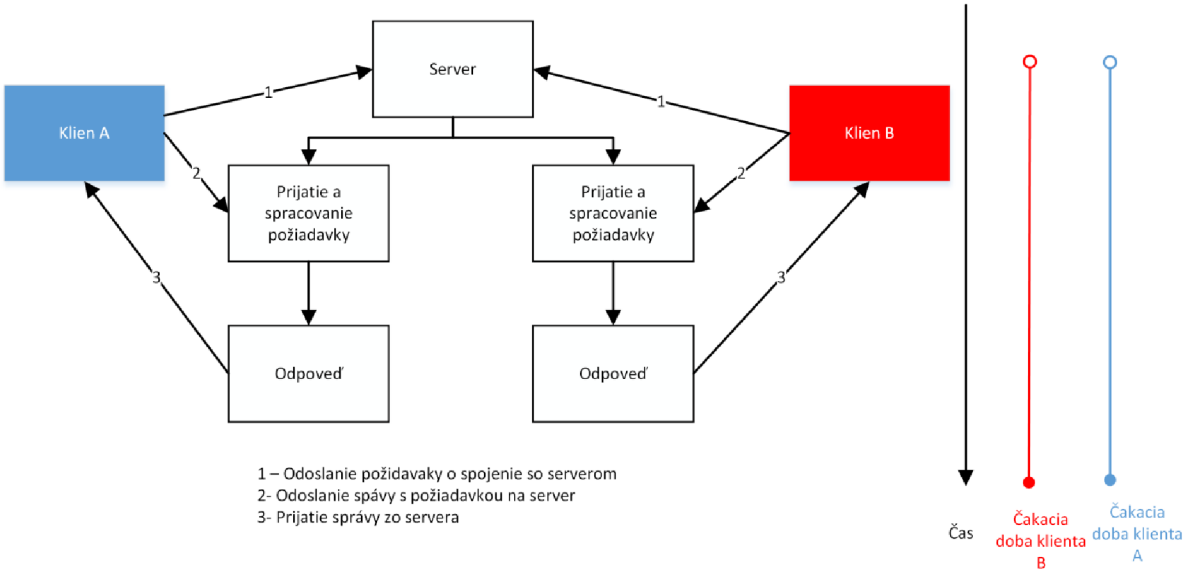
Nevýhoda iteratívneho servera spočíva v tom, že pri príchode dvoch žiadostí o obslúženie v tom istom čase, by jedná so žiadostí musela čakať, kým bude obslúžená prvá žiadosť. To by pri pripočítaní času potrebného na obsluhu samotnej žiadosti spôsobilo vysokú latenciu servera, ktorá je nežiaduca. Koncové zariadenia v domácnosti, hlavne senzory, budú napájané z batérie. Po odoslaní dát cez adaptér na server ostávajú zapnuté a do režimu uspania prechádzajú až v momente, keď dostanú odpoveď s obsahujúcu hodnotu intervalu prebudenia. Vysoká latencia by v tomto prípade spôsobovala vybíjanie batérie počas čakania. Tento problém ilustruje obrázok 3.1.



Obrázok 3.2 - Diagram fungovania iteratívneho servera v čase

Naopak riešenie servera so súbežnou obsluhou eliminuje tento problém. Pre každého pripojeného klienta vytvorí nový proces, v ktorom prebieha jeho obsluha. Tým pádom pri pripojení viacerých klientov v tom istom čase, sú ich požiadavky obsluhované zároveň. Toto riešenie eliminuje

priamy dopad počtu pripojení na dobu obsluhovania klienta. Princíp fungovania a časovej závislosti v tomto druhu servera ilustruje obrázok 3.2.



Obrázok 3.3 - Diagram fungovania servera so súbežnou obsluhou

Riešenie s využitím súbežnej obsluhy klientov takisto prináša určité nevýhody. Vzhľadom na to, že nemáme k dispozícii nekonečné výpočtové zdroje, je počet súbežne obsluhovaných klientov limitovaný. Preto sa ako najlepšie riešenie ukázala kombinácia týchto dvoch prístupov. Pri vysokom počte klientov žiadajúcich obslúženie sa časť klientov obslúži súbežne. Toto množstvo záleží od výpočtovej kapacity systému. Zvyšný klienti sú zaradení do fronty, kde čakajú kým neskončí obsluha jedného alebo viacerých práve obsluhovaných klientov. Pri jej skončení je vybraný zo začiatku fronty počet klientov rovný tomuto číslu. Vybraní klienti sú obslúžení. Počet súbežne obsluhovaných klientov závisí na parametroch systému a iných zdrojov využívaných aplikáciou. Zistenie tohto limitu je predmetom výkonnostných testov.

3.1.5 Procesy a vlákna

Pre vykonávanie súbežných operácii existujú v systéme Linux dve možné riešenia. Sú to procesy a vlákna. V tejto kapitole sa budeme venovať ich popisu a rozdielom. Vykonávanie súbežných operácií nie je v skutočnosti na procesore vykonávané súbežne. Každá z operácií dostáva prístup k procesoru na určitú dobu. Táto doba je veľmi krátka a preto to zdanlivo vyzerá, že operácie sú vykonávané súbežne. Tento princíp sa nazýva prepínanie kontextu. Je vykonávaný operačným systémom a ten rozhoduje, ktorej operácií bude procesor pridelený.

Proces je bežiaca inštancia programu vrátane všetkých hodnôt premenných a stavu. Každý program v Linuxe beží v samostatnom procese. Program môže vytvárať ďalšie procesy. Tie sú nazývané detské procesy. Proces, ktorý ich vytvoril je nazývaný rodičovský. Detské procesy sú v programoch vytvárané volaním funkcie fork. Pri zavolané tejto funkcie je pre detský proces vytvorená kopia operačnej pamäte rodičovského procesu. Z toho vyplýva, že procesy nemajú spoločnú operačnú pamäť. Pre možnosť prístupu k rovnakým dátam je preto nutné explicitne vytvárať priestor nazývaný spoločná pamäť (z anglického výrazu shared memory).

Vlákna sú obdobou procesov. Nazývajú sa aj odľahčené procesy. Pri vytvorení vlákna nie je operačná pamäť kopírovaná ale je naďalej využívaný adresný priestor rodiča. To umožňuje jednoduchšie využívanie dát medzi viacerými vláknami. Táto vlastnosť však vyžaduje vyššiu

obozretnosť pretože môžu vznikať takzvané data races (voľne z angl. preteky o dáta) aj v častiach pamäte, ktoré nie sú zdanlivo spoločné. Najmä kvôli vlastnosti spoločného adresného priestoru sa ukázali vlákna ako vhodnejšie riešenie. Vlákna takisto vyžadujú nižšie réžiu operačného systému pri prepínaní kontextu. Táto vlastnosť sa ukázala ako vhodná, vzhľadom na to, že jedným z požiadaviek na serverovú aplikáciu bolo, čo najrýchlejšie obsluhovanie klientov. Vlákna sú vytvárané inštanciami triedy `thread`. Táto trieda je definovaná v štandardnej knižnici `thread`, jazyka C++ od štandardu ISO C++11[8].

3.1.6 Synchronizácia vlákien

Pri aplikáciách s viacerými vláknami vzniká potreba ich synchronizácie. Vo vývoji pod systémom Linux sú k dispozícií dátové štruktúry `semaphore` a `mutex`.

Štruktúra `semaphore` udržuje počítadlo prechodov. Toto počítadlo sa zvyšuje volaním funkcie `sem_post` a znižuje volaním funkcie `sem_wait`. Hodnota tohto počítadla nikdy neklesá pod hodnotu nula. Pokiaľ má hodnotu nula a je zavolaná funkcia `sem_wait` je vlákno uspané a čaká kým nedôjde k volaniu funkcie `sem_post` iným vláknom.

Štruktúra `mutex` je v podstate špeciálnym druhom štruktúry `semaphore`. Je definovaná v štandardnej knižnici `mutex`, jazyka C++ od štandardu ISO C++11[9]. Metódou na zvyšovanie počítadla je `unlock` a znižovania `lock`. Maximálna hodnota jej počítadla je rovná jednej. To znamená, že inštrukcie medzi volaniami týchto metód môže súčasne vykonávať len jedno vlákno. Ostatné vlákna ostanú pri zavolaní `lock` uspané a čakajú na zavolanie `unlock`.

3.1.7 Komunikácia servera s dvomi stranami

Z požiadaviek na serverovú aplikáciu vyplýva, že musí byť schopná obsluhovať zároveň požiadavky adaptérov a servera obsluhujúceho užívateľov. V časti 3.1.3 popisujúcej komunikačný protokol je vidieť, že správy obsahujú rôzne dáta a majú odlišnú štruktúru. Takisto je komunikácia medzi adaptérom a serverom zabezpečená no komunikácia medzi serverovými aplikáciami nie. Preto by bolo nemožné obsluhovať spojenia od obidvoch strán, keď by boli prijímané cez jeden port. Preto bola aplikácia rozdelená na dve časti podľa druhu požiadaviek, ktoré obsluhujú. Každá z nich prijíma spojenia na inom porte. Využitie oddelených aplikácií nie je možné, lebo aplikácia obsluhujúca požiadavky užívateľov musí posielať správy na adaptér cez spojenia nadviazané adaptérom. Preto každá z týchto častí beží v separátnom vlákne.

3.2 Dátová časť

Táto časť servera slúži na uchovávanie dát o koncových prvkoch a užívateľoch. Jej úlohou je preto sprostredkovávať tieto dáta obidvom stranám komunikačnej časti servera. Jej návrh je preto ovplyvnený potrebami časti servera, ktorá nebola súčasťou tejto práce. V nasledujúcej časti textu popíšem možné návrhy dátovej časti a ich výhody a nevýhody.

Požiadavkou serverovej časti na obsluhu užívateľských zariadení bola dátová štruktúra poskytujúca výstup v jazyku XML. Tato požiadavka vychádza z protokolu spomínanej strany, ktorý je založený na tomto jazyku. Preto boli ako možné riešenie zvolené XML súbory, natívne XML databázy a databázy s možnosťou generovať výstup príkazov v tomto formáte.

Ďalšou požiadavkou bol výhradný prístup k práve editovaným dátam. Potreba výhradného prístupu vychádza z návrhu komunikačnej časti rozdelenej na dve aplikácie a tiež z dôvodu voľby konkurentného servera pre obsluhu klientov. Bez výhradného prístupu k dáta by mohla nastať ich nekonzistencia. Pri súbežnej obsluhe viacerých požiadaviek klientov by mohlo dôjsť k situácii, keď by sa viaceré procesy snažili editovať tie isté prípadne úzko súvisiace dáta. To by spôsobilo, že by sa zmeny vykonané jednou prepísali a tým pádom by došlo k ich strate a vznikol by v domácnosti stav, ktorý užívateľ nedefinoval prípadne by zariadenie užívateľa nezobrazovalo reálny stav.

3.2.1 Riešenie s využitím XML súborov

Prvé riešenie vychádza z použitia XML súborov uložených v súborovom systéme servera. Jeden XML súbor reprezentuje práve jednu domácnosť. Nesie názov podľa čísla adaptéru, cez ktorý domácnosť komunikuje so serverom a nachádza sa v zložke s rovnakým názvom. Obsahom XML súboru sú dáta z koncových zariadení pripojených k danému adaptéru. Súbor je štruktúrovaný podľa týchto zariadení. Súbor sa vytvára pri prvom pripojení adaptéru k serveru. Neobsahuje však ešte žiadne údaje o koncových prvkoch, tie sa tam uložia až pri ich spárovaní s adaptérom. V tom čase ale vystupujú ako neinicializované a inicializujú sa až pri prvej výmene dát. Tieto XML súbory sú aktualizované vždy, keď sa daný adaptér pripojí na server. Ich aktualizácia môže byť vykonaná aj užívateľom z druhej strany. Na serveri bude mať každý užívateľ vytvorenú zložku, v ktorej sa budú nachádzať odkazy na súbory jeho adaptérov.

Dôležitou časťou tohto riešenia je knižnica na manipuláciu s XML súbormi. Funkcie tejto knižnice budú využívané veľmi často preto je nutné aby boli rýchle a nemali vysokú pamäťovú náročnosť. Na výber je relatívne dosť riešení. Po detailnom rozbore vlastností týchto knižníc bola vybraná knižnica pugixml[10].

Najväčším problémom tohto riešenia je zaručenie výhradného prístupu k dátam. Vzhľadom na to, že jeden adaptér môže súčasne vytvoriť viacero pripojení k serveru by sa pokúšalo viacero procesov otvoriť a používať ten istý súbor. Nemuselo by sa jednať o tú istú časť ale pri spracovávaní súborov knižnicami na prácu s XML sa do pamäte načíta celá DOM[11] štruktúra súboru. Tak by vznikla situácia keď by mali dva procesy načítaný v pamäti tú istú neaktuálnu štruktúru súboru. Každý by aktualizoval dáta v inej časti štruktúry. Pri zatváraní súboru by proces, ktorý by skončil skôr uložil svoje dáta. Následne by skončil prácu so súborom aj druhý proces a uložil svoju štruktúru do súboru, čo by znamenalo prepísanie aktuálnych dát starými v časti, ktorú editoval proces 1.

Takisto by to bolo pamäťovo náročne. Preto sa každý súbor otvorí maximálne jeden raz, pričom sa do pamäte načíta celá jeho DOM štruktúra. Pri ďalších pokusoch o jeho otvorenie sa predá len ukazovateľ na jej koreň. Toto riešenie bude vyžadovať štruktúru, do ktorej sa budú ukladať otvorené súbory, ukazovatele na ich koreňový element a počty otvorení. Táto štruktúra bude vyžadovať výhradný prístup. Vyhľadávanie v nej bude musieť byť rýchle, preto bude realizovaná pomocou `hash table` [12]. Pri klesnutí počtu otvorení na hodnotu 0 bude súbor zatvorený a zmeny uložené. Tento prístup je možný vďaka iba pri využití vlákien, ktoré zdierajú jeden pamäťový priestor. Rozdelenie servera na dve aplikácie však niečo takéto neumožňuje. Musel by sa vytvoriť spoločný pamäťový priestor pre tieto aplikácie. Prístup k nemu by musel byť synchronizovaný.

Pôvodný návrh dátovej časti vychádzal z tohto riešenia. Komplikácie, ktoré by nastali nutnosťou synchronizácie sa ukázali ako veľká nevýhoda a preto sa na koniec hľadali iné vhodnejšie alternatívy.

3.2.2 Riešenie s využitím databázy

Ďalším možným riešením ukladania dát je využitie databázy. Toto riešenie prináša aj vyriešenie problému s výhradným prístupom k dátam. O prístup viacerých užívateľov (v našom prípade viacerých aplikácií a ich procesov) sa v databázach stará systém riadenia bázy dát[12]. Nevýhodou využitia databázy je dlhšia doba prístupu k dátam oproti súborovému systému. V nasledujúce časti budú popísané možné riešenia s využitím natívnej XML databázy a databázy s možnosťou výstupu databázových príkazov vo formáte XML.

Natívne XML databázy (ďalej len NXD) majú ako základnú logickú jednotku XML dokument. NXD sa nedajú považovať za nový databázový model. Jedná sa totiž o akúsi nadstavbu pre ukladanie XML dokumentov a zjednodušenie pri práci s nimi. Na fyzickej úrovni ukladania dát dokonca môžu využívať klasické relačné databázy, či súborový systém. NXD vyžaduje vytvorenie modelu. Tento model môže byť vytvorený napríklad v jazyku DOM. Tento model je potom mapovaný na fyzickú vrstvu uloženia dát. Preto je nemožné operovať s dátami na nižších úrovniach bez toho aby nedošlo k porušeniu integrity dát. Z toho vyplýva, že vstup aj výstup NXD je len vo forme XML dokumentov. Pri použití v našom projekte by strana servera komunikujúca s adaptérmi musela zbytočne vytvárať a spracovávať XML dokumenty. To by znamenalo jej spomalenie. Na vytváranie príkazov do NXD sa používa jazyk XPath [14]. Tento jazyk však nebol vyvinutý na prácu s databázovým charakterom dát. To znamená, že má niektoré nedostatky. Preto sa začal vyvíjať jazyk XQuerry [15], ktorý tieto nedostatky dopĺňa. Tento jazyk je stále vo vývoji a preto pre neho ešte neexistujú štandardy. Tento jazyk takisto nie je zapracovaný do všetkých produktov. Najväčšou slabinou NXD je editácia dát. Vo väčšine produktov nie je možné editovať dáta priamo. Je nutné načítať celý XML dokument z databázy editovať ho pomocou nástrojov na editáciu týchto dokumentov a následne uložiť naspäť do databázy. Vzhľadom nato, že editácia dát bude prebiehať pri každom pripojení z domácnosti by to opäť znamenalo spomalenie serveru. Preto sa tento návrh ukázal ako nevhodný.

Databázy s možnosťou výstupu databázových príkazov sú klasické relačné databázy. Obsahujú však nadstavbu, ktorá umožňuje podľa nami zadanej schémy vytvoriť výstup príkazu na získanie dát z databázy vo formáte XML. Výhodou takéhoto prístupu je to, že každá zo serverových častí môže dostávať výstupy vo formáte akom jej vyhovuje a nemusí preto vykonávať nadbytočnú prácu buď pri vytváraní alebo spracovávaní XML dokumentov. Toto riešenie sa ukázalo ako najvhodnejšie. Následne bolo treba vybrať databázový produkt, ktorý sa bude používať. Požiadavkou projektu však bolo nájsť opensource produkt. Preto v úvahu pripadalo PostgreSQL 9.2 alebo MySQL 5.1. MySQL 5.1 však nemá podporu XML zabudovanú preto bol vybraný PostgreSQL 9.2.

3.2.3 Ukladané dáta

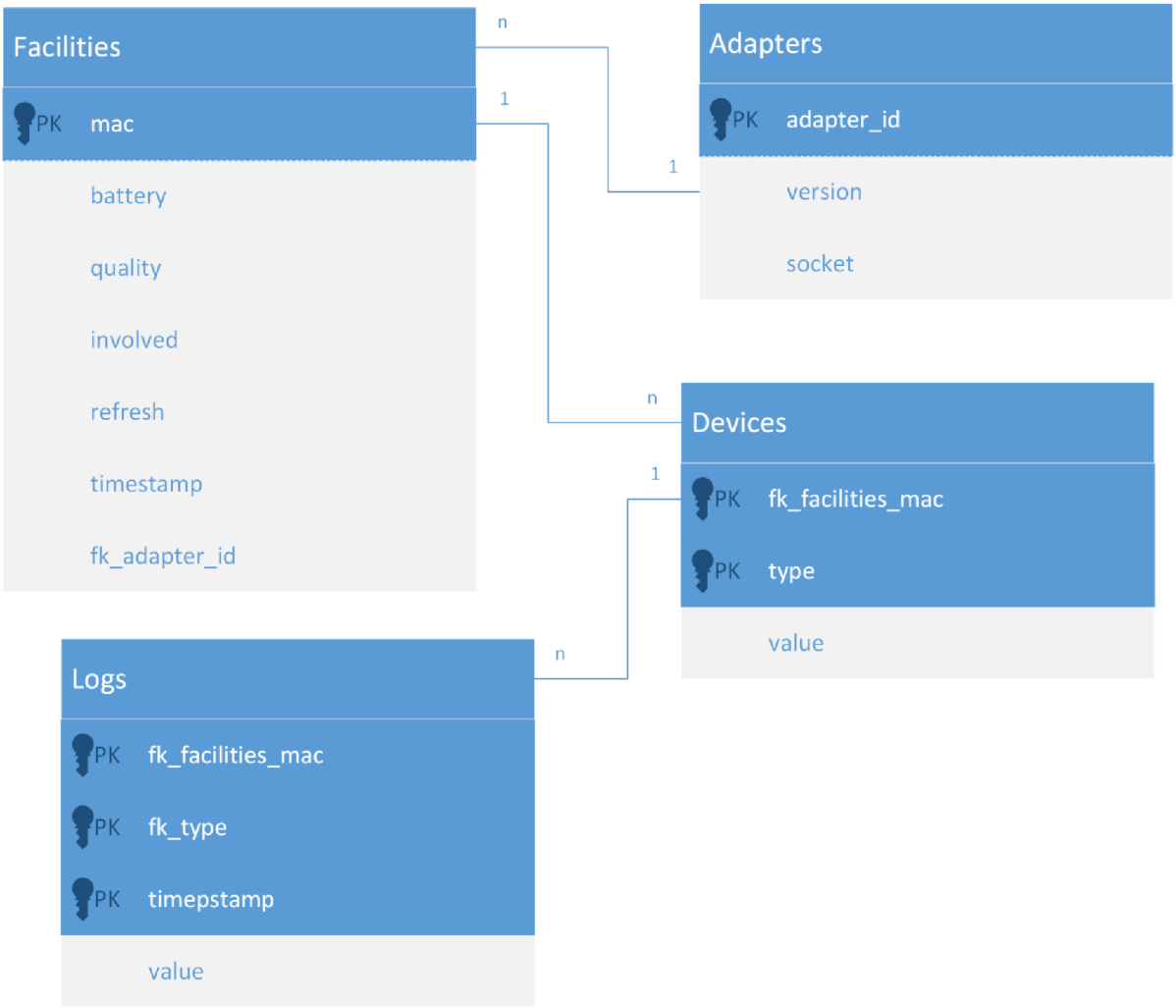
V tejto časti textu budú popísané dáta, ktoré sa budú v dátovej časti uchovávať. V našom systéme vystupujú 3 druhy objektov, o ktorých potrebujeme ukladať dáta. Sú to užívatelia, adaptéry a koncové prvky domácnosti. Údaje o užívateľoch sú pre stranu servera, ktorá je predmetom tejto práce nepodstatné preto sa nimi nebudeme zaoberať.

O adaptéroch potrebujeme uchovávať nasledujúce údaje: jeho identifikáciu a verziu jeho firmvéru. Identifikačný údaj je nutný pre jeho jednoznačnú identifikáciu v rámci systému. Verzia firmvéru sa ukladá pre možnosť jeho vzdialenej aktualizácie.

O koncových prvkoch potrebujeme ukladať nasledujúce dáta: identifikáciu, názov, typ meraných veličín, umiestnenie v domácnosti, dobu za ktorú sa má prebudiť, kvalitu signálu, stav batérie, hodnoty meraných veličín, jeho priradenie k adaptéru a históriu ich hodnôt. Historické hodnoty sa skladajú z identifikácie zariadenia, jeho typu, hodnoty veličiny a časového údaju. Serverová časť na komunikáciu s domácnosťami bude aktualizovať kvalitu signálu, stav batérie a hodnoty meraných veličín. Identifikácia je v aktuálnom návrh tvorená skrátenou MAC adresou zariadenia. Jedná sa o číslo v rozsahu 4 bajtov bez znamienka. Získavať bude časový údaj o ďalšom prebudení.

3.2.4 Návrh databázy

Táto časť textu popisuje spôsob uloženia dát do databázy a jej návrh. Na obrázku 3.3 je zobrazený zjednodušený návrh tabuliek a stĺpcov, s ktorými bude pracovať strana servera komunikujúca s koncovými zariadeniami.



Obrázok 3.4 - Návrh databázy

Tabuľka `Adapters` uchováva dáta o adaptéroch. Jej primárnym kľúčom je `adapter_id` dátového typu `long long integer`. Obsahuje sériové číslo konkrétneho adaptéra. Pri prvom pripojení adaptéru k serveru sa vytvorí v tabuľke nový riadok, do ktorého sa uložia dáta o adaptéri. Pri nasledujúcich pripojeniach sa dáta už len aktualizujú.

Tabuľka `Facilities` uchováva dáta o koncových prvkoch domácnosti. Jej primárny kľúč je stĺpec `mac`, čo je jednoznačná adresa koncového prvku. Stĺpec `fk_adapter_id` slúži na identifikáciu adaptéra ku ktorému je koncový prvok pripojený. Tento stĺpec je aj cudzím kľúčom do tabuľky `Adapters`. Stĺpec `involved` nesie časový údaj prvého pripojenia zariadenia k adaptéru. V stĺpci `timestamp` je uložený čas poslednej aktualizácie dát. Stĺpec `refresh` dĺžku časovej periódy do ďalšieho odoslania dát. Stĺpce `quality` a `battery` obsahujú údaje o kvalite signálu respektíve batérie. Tak isto ako pri adaptéri sa aj pri prvom pripojení koncového zariadenia vytvoria v tabuľke nové riadky patriace k nemu a pri následných pripojeniach sa už len aktualizujú.

Tabuľka `devices` slúži na ukladanie hodnôt jednotlivých pasívnych a aktívnych prvok na koncovom zariadení. Primárny kľúč tvorí kombinácia stĺpcov `fk_ facilities_mac` a `type`. Stĺpec `fk_ facilities_mac` slúži na identifikáciu koncového zariadenia, pod ktoré prvok patrí. V stĺpci type je uchovaný jednoznačný identifikátor koncového prvku v rámci zariadenia. Jeho hodnota Vzniká spojením hodnôt `type` a `offset`, ktorých význam je popísaných v kapitole 3.1.3. Číselná hodnota prvku `offset` sa posunie o 8 bitov doľava. Na tieto pozície sú doplnené nulové bity. Následne sa vykoná logický súčin s číselnou hodnotou prvku `type`. Takto získame nové číslo, ktoré je v rámci zariadenia unikátne. Stále ho však je možné rozložiť naspať. Stĺpec `value` nesie hodnotu daného prvku. Záznam sa v tabuľke vytvára pri prvom poslaní hodnoty prvku zariadením. Potom sa už len aktualizuje.

Do tabuľky `logs` sa ukladajú historické hodnoty koncových prvkov. Jej primárny kľúč sa skladá z 3 častí. Je tvorený stĺpcami `fk_facilities_mac`, `fk_devices_type` a `timestamp`. Dvojica `fk_facilities_mac` a `fk_devices_type` je zhodná s dvojicou `fk_ facilities_mac` a `type` v tabuľke `devices`. Stĺpec `timestamp` slúži na jednoznačnú identifikáciu historickej hodnoty v čase. Posledným stĺpcom tabuľky je `value`. V ňom je uložená konkrétna hodnota. Pre zariadenia s viacerými snímanými veličinami sa pri vkladaní jednej logovacej správy vytvorí počet riadkov totožný s počtom meraných hodnôt.

3.3 Výsledný návrh aplikácie

Zámerom tejto kapitoly je popis prepojenia dátovej časti s komunikačnou. Popisovaný bude návrh častí aplikácie, ktoré priamo nepodliehajú ani jednej z nich, ale ich existencia je podstatná pre chod aplikácie.

3.3.1 Návrh konfigurovania aplikácie

Aplikácia takýchto rozmerov musí byť konfigurovateľná. Existujú v podstate 3 možnosti, ktorými sa táto potreba dá vyriešiť. Prvou z nich je použitie parametrov programu. Tie sa programu predávajú pri spustení za jeho názvom. Táto možnosť je jednoducho realizovateľná, ale jej užívateľskú prívetivosť môžeme považovať za nízku. Druhou z možností je použitie konfiguračného súboru. Jeho výhodou je vyššia prívetivosť pre užívateľa. Jeho štruktúra sa dá postaviť na rôznych šablónach. Jednou z nich je napríklad XML súbor. Ten je vďaka svojej štruktúre čitateľný aj pre užívateľa a jeho štruktúra ľahko modifikovateľná. Ďalšou z výhod je, že konfiguračný súbor stačí vytvoriť raz a môže byť použitý pri opakovanom spúšťaní aplikácie. Poslednou možnosťou je vytvorenie užívateľského rozhrania. Táto možnosť je z pohľadu užívateľskej prívetivosti najlepšia. Vyžaduje však oveľa vyššie úsilie pri tvorbe. Z dôvodu, že aplikácia je vyvíjaná pre konkrétny

projekt a nebude používaná širokou škálou užívateľov, bolo zvolené použitie konfiguračného súboru postaveného formáte XML. Jeho šablónu aj s vysvetlením je možné nájsť v Prílohe A.

3.3.2 Návrh záznamu činnosti aplikácie

Ďalšou z náležitosti rozsiahlejších aplikácií je zaznamenávanie chýb. Pre ich ladenie a odhaľovanie chýb sú ale potrebné aj informácie o ich činnosti. Správy o chybách spolu so záznamom činnosti aplikácie môžeme nazvať logy. V aplikácia s užívateľsky rozhraním je možné tie informácie oznamovať prostredníctvom neho. Pre aplikácie bez neho, môže logovanie prebiehať na štandardný výstup alebo do textového súboru. Nevýhodou štandardného výstupu je obmedzená história logov, ťažké prehľadávanie a celková neprehľadnosť. Túto nevýhodu odstraňuje riešenie s použitím logovacích súborov. Toto riešenie bolo použité pri návrhu logovania v aplikácií serveru.

Maximálna veľkosť tohto súboru je konfigurovateľná v konfiguračnom súbore. Takisto je možné konfigurovať maximálny počet vytvorených súborov. Súbory sú číslované od nula do x, kde x je maximálny počet súborov. Pri dosiahnutí maximálnej veľkosti súboru x sa otvorí súbor s číslom 0 a jeho obsah je prepísaný novými správami.

Logovacie informácie sú rozdelené do 6 kategórií. Jednotlivé úrovne logovania môžu byť vypnuté. Pri zapnutí logovania na určitú úroveň sú logované všetky úrovne s nižším číslom. Zoznam jednotlivých úrovní:

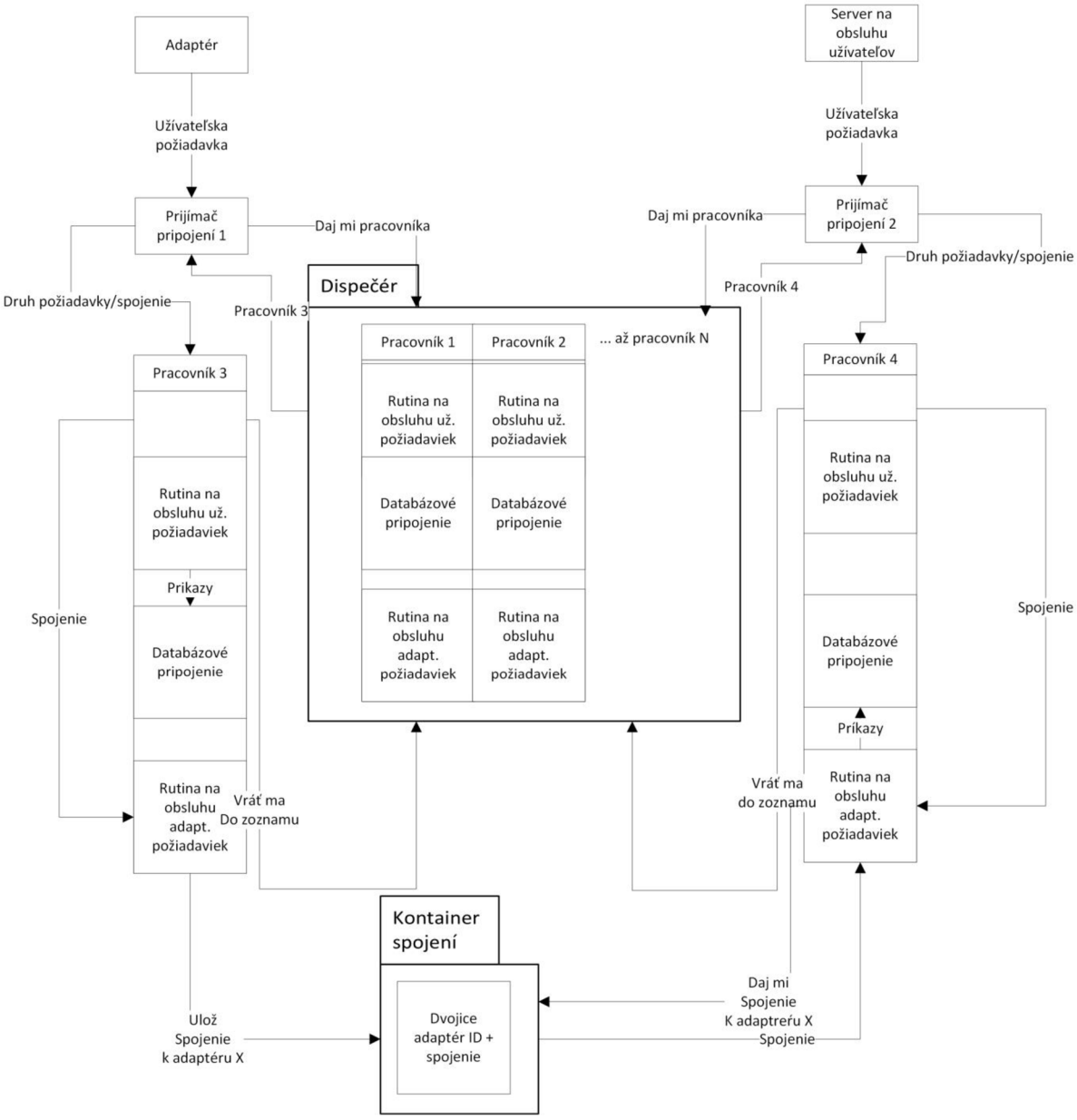
1. FATAL : Správy o chybách, ktoré nastali počas behu servera a viedli k ukončeniu jeho správneho pracovania. Logy tejto úrovne nie je možné vypnúť
2. ERROR : Správy o chybách, ktoré nastali počas behu servera ale nevedú k jeho ukončeniu. Jedná sa napríklad o chyby pri volaní SQL príkazov nad databázou.
3. WARN: Správy upozorňujúce správcu servera na udalosti, ktoré viedli k neobslúženiu klienta. Sú to napríklad situácie, keď prijatá správa nekorešponduje s komunikačným protokolom.
4. INFO: Správy informujúce správcu servera aké operácie server vykonáva.
5. MSG: Správy logujúce podrobné detaily spracovaných správ od klienta.
6. TRACE: Správy logujúce začiatok a koniec vykonávanej operácie. Úlohou týchto správ je zjednodušenie hľadania chýb pri vývoji serverovej aplikácie.

Formát správy tvorí názov aplikácie nasledovaný číslom vlákna. Po ňom je do správy vložený časový údaj, kedy udalosť nastala. Nasleduje textová reprezentácia úrovne správy a samotný text správy ukončený koncom riadku.

Z dôvodu, že logovanie prebieha do textového súboru, dochádza pri ňom k zápisu na disk. To je časovo náročná operácia. Pri súčasnom zápise správ viacerými vláknami by dochádzalo k zmiešaniu obsahu jednotlivých správ, čo by viedlo k ich nečitateľnosti. Preto vyžaduje operácia zápisu výhradný prístup. To v kombinácií s časovou náročnosťou zápisu vedie k spomaleniu práce vlákien aplikácie a spôsobí predĺženie doby obsluhovania klientov. Riešenie tohto problému je presunutie zápisu na disk do separátneho vlákna. Komunikáciu tohto vlákna s ostanými zastrešuje abstraktná štruktúra rada typu FIFO. Zvolená bola z dôvodu zachovania poradia správ. Pre správne fungovanie vyžaduje výhradný prístup. V určitých fázach behu aplikácie nemusí vzniknúť potreba logovania správ aj niekoľko hodín. Aby sa v týchto chvíľach predišlo aktívnemu čakaniu vlákna slúžiaceho na zápis, je počítadlom nezapísaných správ zastavené. Vlákna, ktoré správy do rady vkladajú jeho veľkosť zvyšujú. Pri zvýšení počítadla sa zapisovacie vlákno spustí.

3.3.3 Prepojenie dátovej a komunikačných častí

Ako už bolo vyššie popísané, aplikácia bude využívať na súbežnú obsluhu vlákna. Toto ale prinieslo problém v prepojení na dátovú časť. Knižnice ponúkajúce komunikáciu s PostgreSQL neumožňujú komunikáciu s databázou prostredníctvom jedného pripojenia viacerým vláknam súčasne. Riešenie s vytváraním spojenia pri každom prijatí spojenia serverom je vďaka svojej časovej náročnosti neakceptovateľné. Takisto samotné vytváranie vlákna pre každé nové spojenie s klientom je časovo náročná operácia. Preto sa ako veľmi vhodný ukázal návrhový vzor threadpool [16]. Návrh aplikácie prebral z neho hlavnú myšlienku a to vytvorenie konečnej skupiny vlákien na začiatku programu a ich prideľovanie úlohám počas jeho behu. Z toho, že serverová aplikácia musí obsluhovať len 2 druhy požiadaviek vyplýva, že má len 2 druhy úloh. Preto nie sú vytvárané samostatné vlákna ale takzvaný pracovníci, ktorým je povedané ktorú z týchto úloh majú vykonať. Ilustruje to obrázok 3.5.



Obrázok 3.5 - Model prepojenia aplikácie

Každý pracovník vlastní jedno pripojenie k databáze, cez ktoré s ňou komunikujú obslužné rutiny. Pracovníci sú združovaný takzvaným dispečerom, ktorý ich prideľuje prijímačom spojení. Každý prijímač spojenia beží vo vlastnom vlákne. Dispečer udržuje počítadlo voľných pracovníkov. V momente, keď nie je voľný pracovník a prijímač spojenia požiada o ďalšieho pracovníka, je jeho vlákno zastavené týmto počítadlom do momentu, kým niektorý z pracovníkov neoznámi dispečerovi, že je opäť voľný. Obslužná rutina adaptérov ukladá permanentné spojenia adaptérov do kontajnera spojení. Pri potrebe poslania na adaptér si obslužná rutina užívateľského servera vyžiada toto spojenie z kontajnera spojení. Aby sa prišlo aktívnemu čakaniu voľných robotníkov po spustení svojho vlákna sú zastavený. Po ich priradení sa ich beh spustí a po vykonaní úlohy opäť zastaví.

4 Implementačné detaily

Z návrhu vyplýva, že serverová časť na komunikáciu s domácnosťami obsluhuje dva druhy požiadaviek. Preto môžeme implementáciu rozdeliť na dva logické celky. Prvým je časť aplikácie, ktorá obsluhuje požiadavky prichádzajúce z domácnosti. Druhou je časť, ktorá obsluhuje požiadavky zo strany užívateľského servera a posiela ich na adaptér. Postupne si popíšeme implementáciu najdôležitejších tried a metód týchto dvoch častí.

4.1 Jadro serverovej aplikácie

V tomto odseku budú popísané spoločné časti implementácie obidvoch logických časti aplikácie. Jedná sa o triedu na spracovanie konfigurácie `Config`, triedu zabezpečujúcu logovanie `Loger`, triedu pre rozhranie na komunikáciu s databázou `DBHandler`, triedu `SSLContainer` ukladajúcu trvalé spojenia s domácnosťami, triedu vykonávajúcu súbežnú obsluhu klientov `Worker` a trieda zabezpečujúca vytváranie a prideľovanie inštancii triedy `Worker` nazývanú `WorkerPool`.

4.1.1 Trieda `Config`

Úlohou tejto triedy je načítanie a spracovanie konfiguračného súboru. Pre spracovanie XML štruktúry sa využíva

- Metóda `GetDatabaseProperties` slúži na načítanie údajov o pripojení k databáze.
- Metóda `GetReceiverProperties` načíta konfiguráciu časti servera obsluhujúcej pripojenia od adaptérov.
- Metóda `GetSenderProperties` spracuje detaily konfigurácie pre časť prijímajúcu požiadavky od servera obsluhujúceho užívateľov.
- Metóda `GetLogProperties` spracováva konfiguráciu logovania.
- Metóda `GetCertificatesProperties` načíta cesty k certifikátom potrebným pre správne fungovanie SSL spojenia.
- Metóda `GetCommonProperties` získa konfiguračné údaje spoločné pre celú aplikáciu.

4.1.2 Trieda `Loger`

Táto trieda implementuje návrh logovania popísaný v kapitole 3.3.2. Počítadlo správ je v skutočnosti štruktúra typu `semaphore`. Reprezentuje ju privátna premenná `_msgCounter`. Rada správ je implementovaná pomocou inštancie `QueueMsg` triedy `queue` definovanej v štandardnej knižnici jazyka C++. Výhradný prístup k nej je implementovaný pomocou privátneho objektu `_WriteSemaphore`, ktorý je inštanciou triedy mutex.

Zápis správ do súboru vykonáva metóda `Dequeue`. Táto metóda je volaná z vlákna vytvoreného v konštruktore triedy `Loger`. Telo tejto metódy tvorí cyklus `while`. Prvým príkazom volaným vo vnútri cyklu je `sem_wait` nad počítadlom `_msgCounter`. Ten je v konštruktore triedy

inicializovaný na hodnotu nula. To zabezpečí, že vlákno sa v tomto bode zastaví a čaká, kým nie je zavolaná funkcia `sem_post` vláknom logujúcim správu.

4.1.3 Trieda `DBHandler`

Táto trieda je abstrakciou databázového pripojenia z obrázku 3.5. Jej metódy sú implementáciou správy príkazy. Samotné volanie SQL príkazov a získavanie ich výstupov, zabezpečuje inštancia triedy `session` z knižnice SOCI. Príkazy sú jej predávané v textovej reprezentácii pomocou operátoru << jazyka C++. Chyby databázových príkazov sú naspať do metódy, ktorá ich volala, predávané pomocou výnimiek. Väčšina metód triedy `DBHandler` má jeden vstupný parameter a ním je ukazovateľ na štruktúru typu `tmessage`, ktorá nesie informácie zo spracovaných správ od klientov.

- Metóda `InsertAdapter` ukladá do databázy informácie o adaptéri pri jeho prvom pripojení k serveru.
- Metóda `UpdateAdapter` slúži na aktualizovanie záznamu konkrétneho adaptéru.
- Metóda `InsertSenAct` ukladá informácie koncových zariadení a ich prvokov do databázy pri jeho prvom pripojení k adaptéru a tým pádom aj k systému.
- Metóda `UpdateSenAct` slúži na aktualizáciu dát o koncových zariadenia a ich prvkoch.
- Metóda `LogValue` slúži na uloženie hodnôt poslaných koncovými zariadeniami do histórie hodnôt.
- Metóda `DeleteFacility` odstraňuje z databázy záznam o koncovom zariadení, pri jeho zmazaní z adaptéru užívateľom.
- Metóda `GetWakeUpTime` slúži na získanie času do ďalšieho zobudenia zariadenia. Ako parameter dostáva hodnotu identifikátoru daného zariadenia. Pri chybe prednastavenú hodnotu 5.
- Poslednou dôležitou metódou tohto objektu je metóda `IsInDB`. Jej úlohou je zistiť, či sa konkrétny záznam nachádza v danej tabuľke. Ako parametre dostáva názov tabuľky, názov stĺpca a hodnotu daného záznamu.

4.1.4 Trieda `SSLContainer`

Úlohou tejto triedy je uloženie a spáva dvojíc identifikátor adaptéru a jeho SSL spojenie v pamäti. Implementuje kontajner pripojení z obrázku 3.5. Využíva pri tom pár jednorozmerných polí. V prvom z nich sú uložené identifikátory adaptérov. V druhom sa nachádzajú ukazovatele na SSL spojenia. Princíp ukladania je založený na zhodnosti indexov. Pri ukladaní sa pri prvom vložení uloží do prvého poľa identifikátor adaptéru. Do poľa SSL sa na miesto so zhodným indexom uloží spojenie prináležiace danému adaptéru. Počet uložených prvkov udržuje privátna premenná `size`.

- Metóda `InserSSL` implementuje správu ulož spojenie k adaptéru X. Metóda najprv prejde pole adaptérov a hľadá, či sa daný záznam v ňom už nenachádza. Ak áno uvoľní uložené spojenie z pamäte a hodnotu ukazovateľa prepíše novým. Ak nie identifikátor adaptéru spolu so spojením na index hodnoty `size`, ktorý následne zvýši.
- Metóda `GetSSL` slúži na prístup k spojeniam uloženým v štruktúre. Prechádza pole adaptérov a hľadá daný identifikátor. Keď ho nájde vráti ukazovateľ na SSL spojenie

na príslušnom indexe. V prípade, že hodnota indexu dosiahne hodnotu `size` spojenie v štruktúre nie je. Vtedy vráti ukazovateľ s hodnotou `NULL`.

4.1.5 Trieda `Worker`

Trieda `worker` je abstrakciou takzvaného pracovníka popísaného v kapitole 3.3.3.

- Metóda `Work` slúži na obsluhu klientskych požiadaviek. Jej telo tvorí nekonečný cyklus `while`. Prvým príkazom volaným v tele cyklu je metóda `lock` nad inštanciou `Wait` triedy `mutex`. Takto sa zabezpečí zastavenie vlákna, kým nie je zavolaná z iného vlákna programu metóda `unlock`. K tomuto volaniu dochádza v dvoch prípadoch. Pri potrebe obslúžiť klienta alebo pri ukončovaní programu. Pri ukončovaní programu je nastavená premenná `terminate` na `true`. Vtedy sa ukončí telo metódy a takisto aj beh vlákna. Kontrola tejto premennej je druhým krokom v tele cyklu. Ak má hodnotu `false` pokračuje obslúžením klienta. Následne sa vráti na začiatok cyklu a opäť volá metódu `lock` objektu `Wait`.
- Metóda `Start` slúži na vytvorenie a spustenie vlákna. Vláknu sa ako parameter konštruktoru predá metóda `Work`, ktorej telo má vlákno vykonať. Pred tým sa ešte zavolá metóda `lock` objektu `Wait`, ktorá spôsobí, že sa jeho počítadlo nastaví na 0. Toto vedie k zastaveniu obsluhy v metóde `Work`, popísanému vyššie.
- Metóda `Unlock` implementuje správu druh požiadavky a spojenie z obrázku 3.5. V tele metódy sa volá metóda `unlock` objektu `Wait`. Toto volanie vedie k spusteniu vlákna a pokračovaniu metódy `Work`.
- Metóda `SetTermination` slúži na ukončenie behu vlákna. V jej tele sa nastaví privátna premenná `termination` na `true` a zavolá sa metóda `unlock` objektu `Wait`.

4.1.6 Trieda `WorkerPool`

Trieda `WorkerPool` je abstrakciou pomyselného Dispečera z kapitoly 3.3.3. Je najdôležitejšou triedou programu, pretože ho cely spája. Takisto je aj najnáročnejšia na zdroje. Z týchto dôvodov je v programe, možné inštanciu tejto triedy vytvoriť len raz. Nato je využitý návrhový vzor jedináčik (z angl. singleton) pri jeho implementácií. Tento návrhový vzor zabezpečuje existenciu jedinej inštancie danej triedy v programe [17].

Ukazovatele na inštancie triedy `Worker` sú uložené v jednorozmernom poli. Index posledného voľného pracovníka je uložený v premennej `freeCount`. Vzhľadom na to, že prístup k tomuto poľu potrebujú viaceré vlákna, je vyžadovaný výhradný prístup. Ten je zabezpečený vytvorením inštancie `semaphore` triedy `mutex`. Pred každou manipuláciou s týmto poľom dochádza k volaniu metódy `lock` nad objektom `semaphore`. Po ukončení manipulácie sa volá metóda `unlock`. Počítadlo voľných inštancií triedy `Worker` je implementované štruktúrou `semaphore`.

- Metóda `GetWorker` implementuje správu daj mi pracovníka z obrázku 3.5. Na začiatku metódy sa volá funkcia `sem_wait` nad štruktúrou Sem. Tým sa zníži jeho hodnota. Tu sa z poľa vyberá inštancia triedy Worker s indexom freeCount. Táto premenná je v tomto bode znížená. Nasleduje návrat do miesta volania metódy.
- Metóda `ReturnWorker` slúži ako abstrakcia správy vráť ma do zoznamu z obrázku z obrázku 3.5. To prebieha priradením ukazovateľa na vracanú inštanciu, do poľa na

pozíciu s indexom rovný hodnote premennej `freeCount`. Táto premenná je následne zvýšená o 1. Nasleduje volanie funkcie `sem_post` nad štruktúrou Sem, ktoré spôsobí zvýšenie počítadla voľných inštancií triedy `Worker`.

4.2 Implementácia časti očakávajúceho pripojenia

Abstrakciu tejto časti aplikácie zabezpečuje trieda `AdaServerReceiver`. Prijímanie pripojení klientov je úlohou triedy `ConnectionHandler`, ktorej inštanciu vlastní trieda `AdaServerReceiver`. O obsluhu klientov sa stará trieda `ConnectionServer`. Na spracovanie správ sa používa trieda `ProtocolV1MessageParser`.

4.2.1 Trieda `ConnectionHandler`

Táto trieda implementuje prijímač správ 1 z obrázku 3.5. Na nastavenie prijímania správ slúži metóda `Listen`. Spojenia sa potom prijímané v metóde `ReceiveConnection`.

- Metóda `Listen` na začiatku vytvorí schránku servera. Nastaví komunikáciu na internet, port a rozsah adries, na ktorých bude sever prijímať spojenia. Následne volaním systémovej funkcie `bind` pripojí schránku na daný port. Nasleduje volanie systémovej funkcie `listen` na prepnutie do módu načúvania.
- Metóda `ReceiveConnection` je tvorená nekonečným cyklom. V jeho tele sa volá funkcia `accept` nad schránkou vytvorenou a nastavenou v metóde `Listen`. Funkcia `accept` je blokujúca a preto sa vykonávanie programu zastaví do momentu, kým sa k serveru nepripojí klient. Nasleduje volanie metód implementujúcich správy v obrázku 3.5. Následne sa vráti na začiatok cyklu.

4.2.2 Trieda `ProtocolV1MessageParser`

Úlohou tejto triedy je spracovanie správ prijatých od adaptéru. Implementuje spracovanie správ popísaných v kapitole. Jedná sa prvú verziu protokolu. Táto trieda dedí od abstraktnej triedy `MessageParser`, ktorej účel je zjednodušiť implementáciu a volanie metód tried, ktoré budú spracovávať správy nových verzií komunikačného protokolu. Využíva sa pri tom polymorfizmus. Metódy týchto tried implementujú prehľadávanie XML štruktúry správ pričom získavajú a ukladajú hodnoty jednotlivých atribútov a elementov. Využívajú pri tom metódy implementované v knižnici `pugi_xml`. Na vytvorenie správy nesúcej odpoveď adaptéru slúži metóda `CreateAnswer`.

4.2.3 Trieda `ConnectionServer`

Táto trieda implementuje rutinu na obsluhu adaptéra. Vlastní inštancie tried ProtocolV1MessageParser `DBHandler` a `SSLContainer`.

- Metóda `LoadCertificates` načítava certifikáty potrebné pre zabezpečenú komunikáciu. Využíva pri tom funkcie implementované v knižnici OpenSSL[18]. Načítava certifikát

certifikačnej autority, privátny kľúč a verejný kľúč. Verejný kľúč následne slúži na autentizáciu adaptérom. Výsledkom je potom vytvorenie štruktúry `sslctx`, z ktorej sú vytvárané štruktúry slúžiace na autentizáciu a prijímanie správ.

- Metóda `GetData` slúži na získanie času do ďalšieho zobudenia koncového zariadenia. Vyžíva pri tom metódy triedy `DBHandler IsInDB` pomocou, ktorej zistí, či sa dané zariadenie v databáze už nachádza. Ak nie, vráti prednastavenú hodnotu 5 sekúnd. Ak áno, zavolá metódu `GetWakeUpTime`.
- Metóda `StoreData` ukladá dáta zaslané koncovými prvkami. Pomocou metódy `IsInDB` zistí, či sa daný koncový prvok už v databáze nachádza. Ak áno zavolá metódu `UpdateSenAct`. Ak nie, volá metódu `InsterSenAct`.
- Metóda `HandleConnection` slúži na prijatie správy od klienta. Volá funkciu `SSL_accept`. Tá slúži na ustanovenie zabezpečeného spojenia. Následne prijme správu a pokúsi sa ju spracovať. Pri úspešnom spracovaní, závisí ďalší postup na druhu správy. Pri správach nesúcich dáta sa vytvorí odpoveď. Po odoslaní odpovede sa ukladajú dáta. Táto akcia je odsunutá na koniec z dôvodu časovej náročnosti príkazov do databázy.

  Pri registračnej správe je ukladá do databázy len údaje o adaptéri. Posledným krokom je volanie metódy `InsertSSL` nad inštanciou triedy `SSLContainer`.

4.3 Implementácia programu odosielajúceho dáta

Abstrakciu tejto časti aplikácie zabezpečuje trieda `AdaServerSender` a drží objekt vytvorený z triedy, ktorej úlohou je prijímanie spojení, `Listener`. O obsluhu požiadaviek sa stará trieda `RequestServer`. Na spracovanie správ sa používa trieda `UIServerMessageParser`. Vytvorenie správ zastrešuje trieda `MessageCreator`. Ich poslanie je potom úlohou triedy `Sender`.

4.3.1 Trieda `Listener`

Trieda `Listener` je abstrakciou prijímača 2 z obrázku 3.5. Jej súčasťou sú metódy `Listen` a `HandleConnection`.

- Metóda `Listen` funguje na rovnakom princípe ako metóda `Listen` triedy `ConnectionHandler` popísaná v kapitole 4.2.1.
- Metóda `ReceiveConnection` funguje na rovnakom princípe ako metóda `ReceiveConnection` triedy `ConnectionHanler`, s tým rozdielom, že pri volaní metódy `Unlock` inštancie objektu `Worker`, predáva informáciu o tom, že obsluha bude prebiehať v režime obsluhy požiadavku užívateľského servera.

4.3.2 Trieda `Sender`

Úlohou tejto triedy je posielanie správ na adaptér. Obsahuje jednu metódu nazývanú `Send`. Parametrom tejto metódy je správa a štruktúra SSL, cez ktorú ma byť správa poslaná. Zo štruktúry SSL sa na začiatku metódy získa číslo komunikačnej schránky. Následne sa vykoná kontrola, či v schránke nedošlo k chybe. Ak áno znamená to, že spojenie bolo prerušené a správu nie je možné odoslať. Následne sa číslo schránky v štruktúre SSL zmení na zápornú hodnotu, čím si program zapamätá, že je táto schránka nepoužiteľná. V prípade, že je všetko v poriadku je správa odoslaná

volaním funkcie `SSL_write`. Pri úspešnom odoslaní správy metóda vráti hodnotu `true`. V opačnom prípade `false`.

4.3.3 Trieda `UIServerMessageParser`

Táto trieda slúži na abstrakciu spracovania správy od serveru spracovávajúceho požiadavky užívateľov. Implementuje metódy prechádzajúce XML správu pomocou knižnice `pugi_xml`. Jedinou jej verejnou metódou je metóda `ParseMessage`. Ta následne volá privátne metódy implementujúce spracovanie jednotlivých prvkov a atribútov správy.

4.3.4 Trieda `MessageCreator`

Účelom triedy `MessageCreator` a jej metód je na základe typu správy od užívateľského servera vytvoriť príslušnú správu reprezentujúcu požiadavku pre adaptér. Na vytvorenie správ využíva metódy a triedy z knižnice `pugi_xml`. Metóda `CreateDeleteMessage` slúži na vytvorenie mazacej správy. Metóda `CreateListenMessage` implementuje vytvorenie správy, ktorá uvedie adaptér do režimu párovania. Pomocou metódy `CreateSwitchMessage` je vytváraná správa na prepnutie aktuátoru. Stavba jednotlivých správ je popísaná v kapitole 3.1.3.

4.3.5 Trieda `RequestServer`

Trieda `RequestServer` je abstrakciou rutiny na spracovanie požiadaviek užívateľského servera z obrázku 3.5. Vlastní inštancie tried `DBHandler`, `MessageCreator`, `Sender` a `UIServerMessageParser`. Najdôležitejšou metódou je metóda `HandleRequest`, ktorá združuje celú logiku. Táto metóda je volaná v metóde `Work` inštancie triedy `Worker`. Na jej začiatku sa prečíta správa z komunikačnej schránky. Využíva pri tom funkciu `read`. Následne volá metódu `ParseMessage` inštancie triedy `UIServerMessageParser`. Podľa druhu požiadavku sa volá príslušná metóda inštancie triedy `MessageCreator` na vytvorenie správy pre adaptér. Ďalej sa volá metóda `GetSSL` nad inštanciou triedy `SSLContainer`, ako parameter jej predá identifikátor adaptéru získaný zo správy. Táto inštancia je používaná aj metódou `HandleConnection` triedy `ConnectionServer` v časti aplikácie obsluhujúcej adaptéri. V prípade, že všetko prebehne v poriadku, sa volá metóda `Send` inštancie triedy `Sender`, ktorá odošle správu adaptéru. Ak sa jedná o požiadavku na zmazanie koncového prvku volá sa ešte metóda `DeleteFacility` inštancie triedy `DBHandler`. Toto volanie je podmienené úspechom odoslania správy na adaptér. Poslednou akciou je odoslanie správy o úspechu alebo neúspechu posielania správy užívateľskému serveru, ktorý ju spropaguje užívateľovi.

4.4 Previazanie jednotlivých častí s jadrom aplikácie

Táto podkapitola popisuje previazanie inštancií tried popísaných v ostatných podkapitolách. Základom aplikácie je inštancia triedy `WorkerPool`. Ukazovateľ na ňu obsahujú objekty vytvorené z tried `Listener` a `ConnectionHandler`. Každá inštancia triedy `Worker` obsahuje jeden objekt

triedy `ConnectionServer` a jeden objekt triedy `RequestServer`. Objekty tried `ConnectionServer` a `RequestServer` obsahujú ukazovateľ na rovnakú inštanciu triedy `SSLContainer`. Každá vlastní svoj objekt triedy `DBHandler` ale inštancia triedy `session`, ktoré vlastnia tieto objekty je rovnaká. Logovanie správ je vykonávané do dvoch súborov podľa častí serverovej aplikácie. Preto v rámci programu existujú dva objekty triedy `Loger`. Objekty vytvorené, ako inštancie tried jednotlivých častí, vlastnia ukazovateľ na objekt zodpovedný za logovanie ich časti aplikácie. Objekty, ktoré sú spoločné majú ukazovateľ na obidva tieto objekty. Objekt použitý na logovanie sa volí podľa časti, z ktorej bola metóda volaná. Všetky ukazovatele sú predávané ako parametre konštruktorov tried.

5 Testovanie

Táto kapitola popisuje testovanie výslednej aplikácie. Úlohou testovania bolo overenie správnej funkcionality servera a jeho odolnosti voči záťaži.

5.1 Testovanie funkcionality

Účelom tohto testovania bolo zaručiť, že aplikácia spĺňa požiadavky na funkcionalitu. Boli vykonávané počas celého procesu implementácie. Testy môžeme rozdeliť do dvoch kategórií podľa ich účelu. Pozitívne testy overovali správanie serverovej aplikácie pri očakávaných situáciách a vstupoch. Negatívne testy sa zameriavali na neočakávané situácie a nesprávne vstupy.

Za vstupy aplikácie sú považované správy prijímané od klienta. Testovanie očakávaných vstupov bolo vykonávané emulátorom adaptéra a koncových zariadení. V neskoršej fáze aj reálnym adaptérom. Overovanie korektného spracovania vstupov prebiehalo kontrolou dát ukladaných do databázy, overovaním správ, ktoré posielal server ako odpoveď a kontrolou logovacích výpisov. Správny postup servera bol overovaný správny poradím volania metód kontrolovaných v logovacích výpisoch. Takto bolo overené spracovanie správ, správnosť databázových príkazov, predávanie parametrov medzi metódami a objektmi. Overila sa tak aj korektnosť logiky fungovania aplikácie.

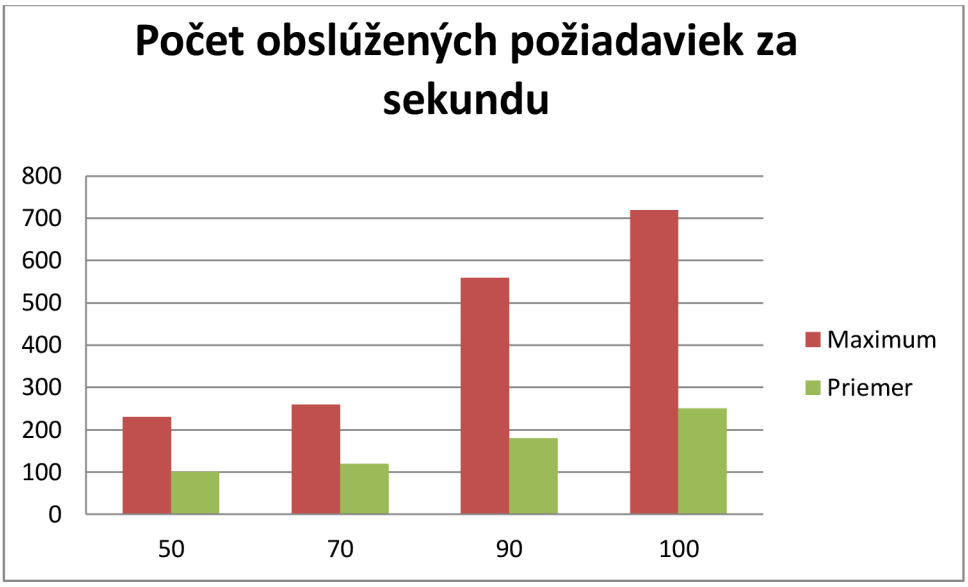
Testovanie neočakávaných vstupov bolo vykonávané jednoduchou aplikáciou napísanou v jazyku C++, ktorá posielala na server správy. Tieto správy obsahovali rôzne chyby alebo nezodpovedali protokolu. Medzi chybami môžeme rozumieť hodnoty presahujúce maximálne rozsahy, správy z nesprávnou XML štruktúrou a podobne. Testovanie neočakávaných situácií malo za účel overiť správne spracovávanie chybových udalostí. Jednalo sa chyby pri spúšťaní serverovej aplikácie. Medzi ne patrí neschopnosť sa pripojiť k databáze, obsadené číslo portu, nedostatočné práva na súbor pre logovanie, neexistujúci alebo nesprávny súbor s konfiguráciu a podobne. Ďalšími situáciami bolo neočakávané správanie klientov, to znamená prerušenie spojenia počas komunikácie, pokus o odoslanie správy na neexistujúci adaptér alebo adaptér, ktorý prerušil permanentné spojenie. Pomocou programu valgrind bola sledovaná správna práca s pamäťou a jej uvoľňovanie, ako počas behu, tak aj pri vypínaní serverovej aplikácie. Poslednou fázou tohto testovania boli testy dlhého behu, keď aplikácia bežala bez prerušenia niekoľko týždňov pričom boli vykonávane obidva druhy testov.

5.2 Záťažové testy

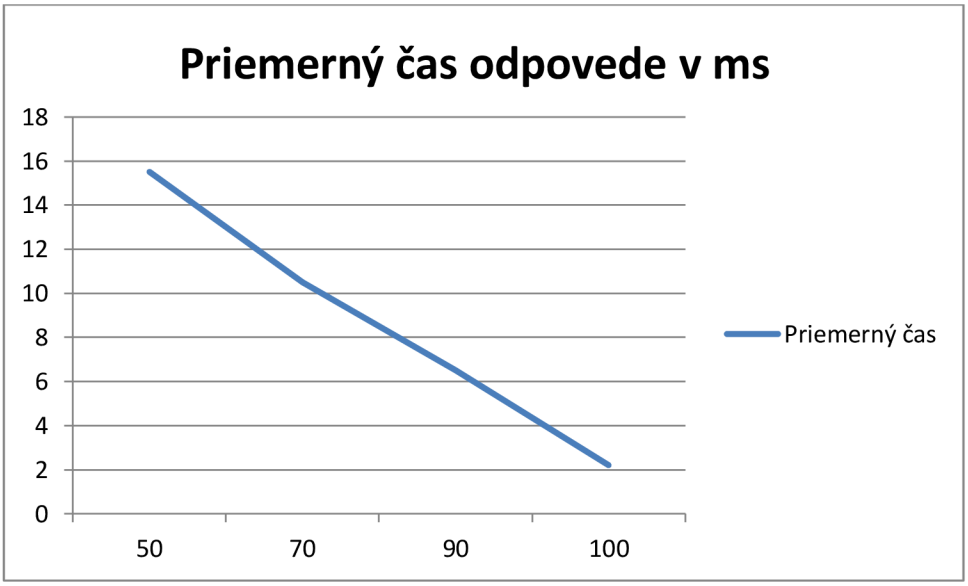
Úlohou záťažových testov bolo overenie správnej funkcionality aj pod vysokou záťažou. Ďalším z účelov bolo nájsť limity serverovej aplikácie. Konkrétne maximálny počet požiadaviek, ktoré je aplikácia schopná obslúžiť za sekundu. Záťažové testy boli vykonávané pomocou emulátoru koncových prvokov. Výkonnostne testy boli vykonávane na stroji s procesorom Intel(R) Xeon(R) CPU E5410 @ 2.33GHz a operačnou pamäťou o veľkosti 12 gigabajtov. Testované boli dva scenáre. Úlohou prvého scenára bolo zistiť ako sa na výkone aplikácie prejaví zvyšovanie počtu pripojení k databáze. Predmetom druhého scenáru bolo zistiť pri optimálnom počte spojení limit počtu požiadaviek za sekundu, ktoré je server schopný obslúžiť.

5.2.1 Scenár 1

Zo špecifikácie PostgreSQL databázy vyplýva, že pri základných nastaveniach je možné vytvoriť maximálne 100 pripojení k databáze. Jednotlivé počty vytvorených pripojení boli 50,70,90 a 100. Emulátor adaptéra a koncových prvkov bol nastavený tak, že emuloval 500 adaptérov s jedným koncovým prvkom. Interval posielania dát zariadeniami bol generovaný pseudonáhodne v rozmedzí 5 až 20 sekúnd. Výsledky meraní sú zobrazené v grafoch 5.1 a 5.2.



Graf 5.1 - Počet obslúžených požiadaviek



Graf 5.2 - Priemerný čas odpovede

Výsledky meraní ukázali, že schopnosť serveru obsluhovať viac požiadaviek narastá v závislosti na počte vlákien slúžiacich na obsluhu. Maximálny počet obslúžených požiadaviek bol dosiahnutý pri registrácii adaptérov. Priemerný počet obslúžených požiadaviek nekopíruje rast maximálneho počtu aj z dôvodu, že pri vyšších počtoch server stíhal odsluhovať požiadavky rýchlejšie a preto vznikali časové úseky, keď klientov neobsluhoval vôbec, lebo nemal nahromadené

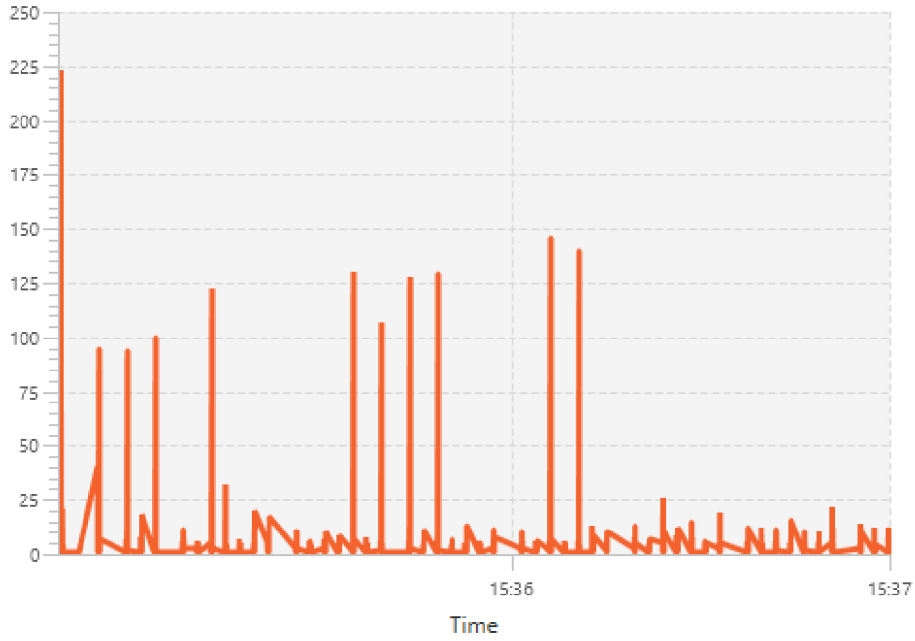
požiadavky. Toto dokazuje aj priebeh času potrebného na obsluhu klientov. Čas sa výrazne skracoval s vyšším počtom vlákien.

5.2.2 Scenár 2

Na základe Scenára 1 bol určený ako optimálny počet spojení s databázou a tým pádom aj vlákien 100. Počas tohto scenáru bol počet adaptérov nastavený na 1000. Premennou časťou scenára bol počet koncových zariadení pripojených k nim. Ten bol 1,2 a 4.

Pri 1000 adaptéroch s jedným senzorom serverová aplikácia stáhala vybavovať všetky požiadavky klientov bez výrazných časových oneskorení. Priemerná doba obsluhy požiadavky sa nachádzala v okolí 5 milisekúnd.

Počas behu úlohy s 1000 adaptérmi s dvomi koncovými zariadeniami sa sporadicky začali objavovať neobslúžení klienti, ktorých spojenia boli aplikáciou odmietnuté. K tomuto javu dochádzalo vo chvíľach keď bol na server prijatý väčší zhluk správ v jednom čase. Je to spôsobené funkciou `accept`, ktorá je zodpovedná za prijímanie správ. Ako parameter sa jej predáva počet prijatých spojení čakajúcich na obslúžene. Ten bol nastavený na 100, aby sa pre každé obslužné vlákno udržiavalo maximálne jedno čakajúce spojenie. Počas testu sa takisto začali objavovať požiadavky, ktorých obslužný čas výrazne presahoval priemer. Toto je ilustrované na grafe 3.3 vytvorenom aplikáciou emulátoru.



Priebeh úlohy s 1000 adaptérmi po štyroch senzoroch viedol k pádu aplikácie. Počas prvých minút behu sa začali objavovať neobslúžení klienti. Tento počet mal s rastúcim časom stúpajúcu tendenciu. Po pár minútach sa začali objavovať chybové hrášky v logovacom súbore, spojené s chybami sieťových funkcií. Ich výsledkom bolo zrútenie serverovej aplikácie. Treba si uvedomiť, že pri tejto úlohe bolo generovaných v priemere viac než 350 požiadaviek za sekundu a v nárazoch približne 1000. Výsledkom je zistenie, že serverová aplikácia je schopná krátkodobo obslúžiť aj takýto vysoký počet klientov ale doba nesmie prekročiť rády sekúnd.

6 Záver

Cieľom tejto bakalárskej aplikácie bolo vytvoriť serverovú aplikáciu komunikujúcu s domácnosťami v rámci projektu inteligentnej domácnosti. Aby som mohol splniť tento cieľ, musel som podrobne naštudovať architektúru tohto systému. Taktiež som sa musel zoznámiť s princípmi tvorenia serverových aplikácií a sieťovými technológiami používanými v systému Linux. Na základe týchto znalostí som navrhol a následne implementoval aplikáciu so zameraním na výkonnosť a efektivitu vo využívaní prostriedkov operačného systému. V rámci tejto práce som sa takisto podieľal na návrhu časti systému zastrešujúcu komunikáciu medzi serverom a adaptérom. Podmienkou úspešnej implementácie bolo naštudovanie spôsobu tvorby objektovo orientovaných návrhov aplikácií v jazyku C++.

Výsledná aplikácia je schopná prijímať a uchovávať dáta z domácnosti. Druhou úlohou, ktorú plní je sprostredkovávanie príkazov užívateľov do domácností. Ako nadstavbu ponúka užívateľsky zrozumiteľné logovanie vykonávaných akcií a jednoduchú konfiguráciu. Tieto časti aplikácie sú používané aj inými časťami systému, čo dokazuje ich využiteľnosť.

Poslednou ale nemenej dôležitou časťou práce bolo testovanie aplikácie. Testovanie dokázalo, že aplikácia spĺňa svoj účel. Výsledky testov takisto ukázali, že boli splnené predsavzatia stanovené na výkonnosť a šetrenie zdrojov operačného systému.

Výsledok mojej práce sa stal neodlúčiteľnou súčasťou projektu inteligentnej domácnosti vyvíjanej na FIT VUT. V rámci tohto sýtemu je nasadený vo vývojovej verzií a takisto v takzvanej produkčnej verzií, ktorá simuluje reálne nasadenie systému.

Vývoj aplikácie je úzko spätý s rozvojom celého systému. Jeho postupný vývoj a rast bude prinášať nové požiadavky na serverovú aplikáciu, ktoré bude nutné zapracovať. Ďalšou z možností rozvoja práce je optimalizácia výkonu. Tá spočíva v reimplementácii štruktúry uchovávajúcej permanentné spojenia. Zámerom by mala byť možnosť efektívneho vyhľadávania v nej. Zaujímavou ideou pri zvyšovaní výkonu aplikácie by bolo preskúmanie dynamického prideľovania databázových pripojení, ktoré by v konečnom dôsledku umožnilo vytvorenie vyššieho počtu vlákien, slúžiacich na obsluhu klientov. Aplikácia takisto z dôvodu jednoduchšej možnosti testovania neoveruje identitu adaptérov na základe ich certifikátov (viď 3.1.2.). Implementácia tejto kontroly je nevyhnutná pre bezpečnosť systému.

Literatúra

[1] Internet of Things. 2007. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, Naposledy modifikované 16:19, 8.4.2015. [cit. 2015-05-04]. Dostupné na: http://en.wikipedia.org/wiki/Internet_of_Things#Original_definition

[2] Network address translation. 2009. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, Naposledy modifikované 16:19, 15. 3. 2013. [cit. 2015-02-04]. Dostupné na: http://sk.wikipedia.org/wiki/Network_address_translation

[3] Virtuální privátní síť. 2009. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, Naposledy modifikované 16:19, 12.9.2014. [cit. 2015-02-04]. Dostupné na: http://cs.wikipedia.org/wiki/Virtuální_privátní_síť

[4] SSL protokol. *Igloonet, s.r.o* [on-line]. [cit. 2015-02-08]. Dostupné na : https://www.ssl-certifikaty.cz/o-certifikatech/ssl-protokol/

[5] OPPLIGER, Rolf. *SSL and TLS: theory and practice*. Boston: Artech House, c2009, xxi, 257 p. Artech House information security and privacy series.

[6] Extensible Markup Language (XML) 1.0 (Fifth Edition). 2008.*World Wide Web Consortium (W3C)* [on-line]. Naposledy modifikované 7. 2.2013. [cit. 2015-02-04]. Dostupné na: http://www.w3.org/TR/xml/

[7] Studijní opora předmětu ISA, *FIT VUT v Brňe* 10.10.2011. [cit. 2015-02-04]

[8] Thread support library. *Cppreference* [on-line]. [cit. 2015-02-08]. Naposledy modifikované 12:24, 7.9.2014. http://en.cppreference.com/w/cpp/thread

[9] C++ Standard Library header files. *Cppreference* [on-line]. Naposledy modifikované 8:41, 21.3.2015. [cit. 2015-02-08]. Dostupné na: http://en.cppreference.com/w/cpp/header

[10] Pugi XML library documentation. *pugixml* [on-line]. Naposledy modifikované 20:49:27, 10.4.2015. [cit. 2015-05-05]. Dostupné na: http://pugixml.org/docs/manual.html

[11] Document Object Model *Wikipedia: the free encyclopedia* [on-line]. Naposledy modifikované 11:40, 8.4.2015. [cit. 2015-10-04]. Dostupné na: http://en.wikipedia.org/wiki/Document_Object_Model

[12] Hash Table. 2001. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, Naposledy modifikované 7.2.2015 [cit. 2015-02-04] Dostupné na: http://en.wikipedia.org/wiki/Hash_table

[13] Database Management system. *PCMag Digital Group* [on-line]. [cit. 2015-02-08]. Dostupné na: http://www.pcmag.com/encyclopedia/term/40952/dbms

[14] XML Path Language (XPath). 2008.*World Wide Web Consortium (W3C)* [on-line]. [cit. 2015-02-04]. Naposledy modifikované 7. Február 2013. Dostupné na: http://www.w3.org/TR/xpath/

[15] XQuery 1.0: An XML Query Language (Second Edition). 2014.*World Wide Web Consortium (W3C)* [on-line]. [cit. 2015-02-04]. Naposledy modifikované 3.1.2011. Dostupné na: http://www.w3.org/TR/xquery/

[16] GARG, Rajat P a Ilya A SHARAPOV. *Techniques for optimizing applications: high performance computing*. Upper Saddle River, NJ: Prentice Hall PTR, c2002, xliii, 616 p. ISBN 0130934763.

[17] GAMMA, Erich. *Design patterns elements of reusable object-oriented software*. Reading: Addison-Wesley, c1995, xv, 395 s. Addison-Wesley professional computing series. ISBN 0201633612.

[18] SSL - OpenSSL SSL/TLS library. *OpenSSL project* [on-line]. [cit. 2015-02-08]. Dostupné na: https://www.openssl.org/docs/ssl/ssl.html

Zoznam príloh

1. Príloha A - Šablóna konfiguračného súboru
2. Príloha B - Ukážka logovacieho súboru
3. Príloha C - UML diagram tried

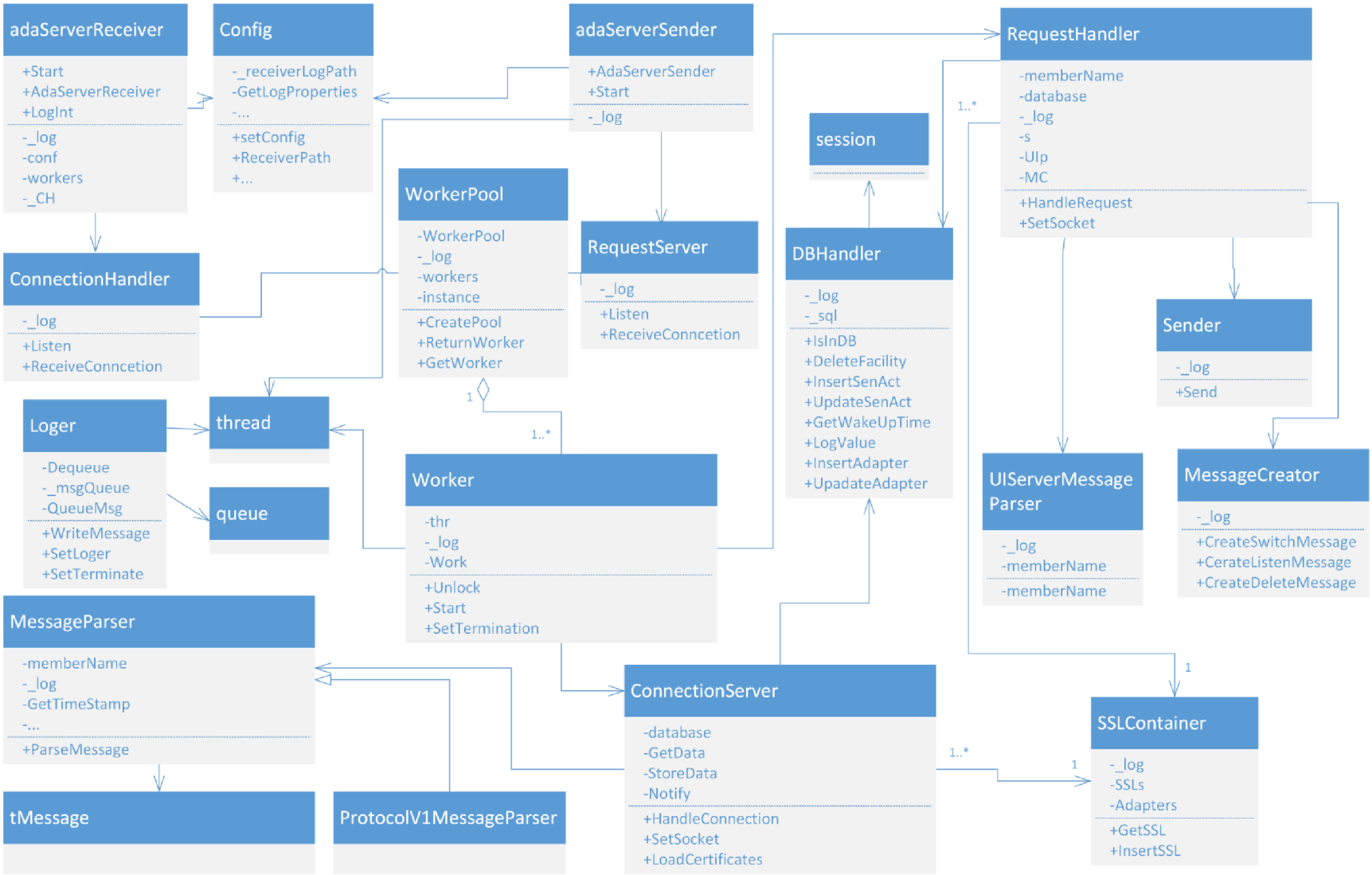
Príloha A

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <Database>
    <Name>home6</Name> <!—Názov databázy-->
    <User></User> <!-- DB administrátor databázy DB-->
    <Password></Password> <!—heslo (ak bude nevyplnené program si ho pri starte
vyziada) -->
    <ConnectionsCount>15</ConnectionsCount> <!—Počet spojení k DB -->
  </Database>
  <Common>  <!—spoločné nastavenia aplikácie -->
    <Certificates> <!—konfigurácia certifikátov -->
      <KeyFile>/etc/openvpn/server.key</KeyFile> <!--Cesta ku priv. kľúču-->
      <CrtFile>/etc/openvpn/server.crt</CrtFile> <!—Cesta k verejnému kľúcu-->
      <CACrtFile>/etc/openvpn/ca.crt</CACrtFile> <!—Cest k CA certifikátu -->
    </Certificates>
    <Mode>DEAMON</Mode> <!—Mód spustenia-->
  </Common>
  <Sender> <!—konfigurácia časti posielajúcej dáta-->
    <LogConfig> <!—konfigurácia logovania -->
      <Level>7</Level>
      <FileNaming>ada_server_sender</FileNaming> <!—Názov súborov -->
      <FilesCount>5</FilesCount> <!—DĺžKa histórie súborov -->
      <MaxFileSize>100</MaxFileSize> <!-- Size of log file in MB -->
      <LogPath>/home/xblaho03/logs/</LogPath> <!—Zložka logovania -->
    </LogConfig>
    <Port>7081</Port>  <!—port pre očakávanie spojení -->
  </Sender>
  <Receiver> <!-- konfigurácia časti prijimajúcej dáta -->
    <Port>7080</Port> <!-- port pre očakávanie spojení -->
    <ConnectionTimeOut>10</ConnectionTimeOut> <!—časový limit na prijatie správy --
>
    <LogConfig> <!—Viď v časti Sender-->
      <Level>7</Level>
      <FileNaming>ada_server_receiver</FileNaming>
      <FilesCount>5</FilesCount>
      <MaxFileSize>100</MaxFileSize>
      <LogPath>/home/xblaho03/logs/</LogPath>
    </LogConfig>
    </Receiver>
 </configuration>
```

Príloha B

RECEIVER [1507629376] (15/05/16_14:15:48.357012) TRACE : Entering ProtocolV1MessageParser::GetState
RECEIVER [1507629376] (15/05/16_14:15:48.357015) MSG : STATE :1
RECEIVER [1507629376] (15/05/16_14:15:48.357016) TRACE : Exiting ProtocolV1MessageParser::GetState
RECEIVER [1507629376] (15/05/16_14:15:48.357017) TRACE : Entering ProtocolV1MessageParser::GetTimeStamp
RECEIVER [1507629376] (15/05/16_14:15:48.357020) MSG : Timestamp int val:0 UTC
RECEIVER [1507629376] (15/05/16_14:15:48.357029) MSG : Timestamp :2015-05-16 12:24:07 UTC
RECEIVER [1507629376] (15/05/16_14:15:48.357030) TRACE : Exiting ProtocolV1MessageParser::GetTimeStamp
RECEIVER [1507629376] (15/05/16_14:15:48.357032) TRACE : Exiting ProtocolV1MessageParser::parseMessage
RECEIVER [1507629376] (15/05/16_14:15:48.357033) TRACE : Entering MessageParser::ReturnMessage
RECEIVER [1507629376] (15/05/16_14:15:48.357034) TRACE : Exiting MessageParser::ReturnMessage
RECEIVER [1507629376] (15/05/16_14:15:48.357037) TRACE : Entering DBHandler::IsInDB
RECEIVER [1507629376] (15/05/16_14:15:48.357038) TRACE : params : adapter_id 1085 adapters
RECEIVER [1507629376] (15/05/16_14:15:48.357040) TRACE : select count(*)adapter_id from adapters where adapter_id = 1085;
RECEIVER [1434200384] (15/05/16_14:15:48.357074) TRACE : Entering ProtocolV1MessageParser::Constructor
RECEIVER [1434200384] (15/05/16_14:15:48.357075) TRACE : Exiting ProtocolV1MessageParser::Constructor
RECEIVER [1434200384] (15/05/16_14:15:48.357077) TRACE : Entering ProtocolV1MessageParser::parseMessage
RECEIVER [1339791680] (15/05/16_14:15:48.357085) MSG : Timestamp int val:0 UTC
RECEIVER [1339791680] (15/05/16_14:15:48.357094) MSG : Timestamp :2015-05-16 12:24:07 UTC
RECEIVER [1339791680] (15/05/16_14:15:48.357095) TRACE : Exiting ProtocolV1MessageParser::GetTimeStamp
RECEIVER [1339791680] (15/05/16_14:15:48.357096) TRACE : Exiting ProtocolV1MessageParser::parseMessage
RECEIVER [1339791680] (15/05/16_14:15:48.357098) TRACE : Entering MessageParser::ReturnMessage
RECEIVER [1339791680] (15/05/16_14:15:48.357099) TRACE : Exiting MessageParser::ReturnMessage
RECEIVER [1444690240] (15/05/16_14:15:48.357104) MSG : Timestamp int val:0 UTC
RECEIVER [1339791680] (15/05/16_14:15:48.357107) TRACE : Entering DBHandler::IsInDB
RECEIVER [1339791680] (15/05/16_14:15:48.357108) TRACE : params : adapter_id 1215 adapters
RECEIVER [1339791680] (15/05/16_14:15:48.357111) TRACE : select count(*)adapter_id from adapters where adapter_id = 1215;
RECEIVER [1444690240] (15/05/16_14:15:48.357115) MSG : Timestamp :2015-05-16 12:24:07 UTC
RECEIVER [1444690240] (15/05/16_14:15:48.357117) TRACE : Exiting ProtocolV1MessageParser::GetTimeStamp
RECEIVER [1444690240] (15/05/16_14:15:48.357118) TRACE : Exiting ProtocolV1MessageParser::parseMessage
RECEIVER [1444690240] (15/05/16_14:15:48.357119) TRACE : Entering MessageParser::ReturnMessage
RECEIVER [1444690240] (15/05/16_14:15:48.357120) TRACE : Exiting MessageParser::ReturnMessage

Príloha C


adaServerReceiver
+Start
+AdaServerReceiver
+LogInt
-_log
-conf
-workers
-_CH
Config
-_receiverLogPath
-GetLogProperties
-...
+setConfig
+ReceiverPath
+...
adaServerSender
+AdaServerSender
+Start
-_log
RequestHandler
-memberName
-database
-_log
-s
-UIp
-MC
+HandleRequest
+SetSocket
session
ConnectionHandler
-_log
+Listen
+ReceiveConncetion
WorkerPool
-WorkerPool
-_log
-workers
-instance
+CreatePool
+ReturnWorker
+GetWorker
RequestServer
-_log
+Listen
+ReceiveConncetion
DBHandler
-_log
-_sql
+IsInDB
+DeleteFacility
+InsertSenAct
+UpdateSenAct
+GetWakeUpTime
+LogValue
+InsertAdapter
+UpadateAdapter
Sender
-_log
+Send
Loger
-Dequeue
-_msgQueue
-QueueMsg
+WriteMessage
+SetLoger
+SetTerminate
thread
queue
Worker
-thr
-_log
-Work
+Unlock
+Start
+SetTermination
UIServerMessage Parser
-_log
-memberName
-memberName
MessageCreator
-_log
+CreateSwitchMessage
+CerateListenMessage
+CreateDeleteMessage
MessageParser
-memberName
-_log
-GetTimeStamp
-...
+ParseMessage
ConnectionServer
-database
-GetData
-StoreData
-Notify
+HandleConnection
+SetSocket
+LoadCertificates
SSLContainer
-_log
-SSLs
-Adapters
+GetSSL
+InsertSSL
tMessage
ProtocolV1MessageParser
1
1..*
1..*
1
1..*
1