VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

# MODULÁRNÍ PROSTŘEDÍ PRO ZPRACOVÁNÍ SENZORICKÝCH DAT

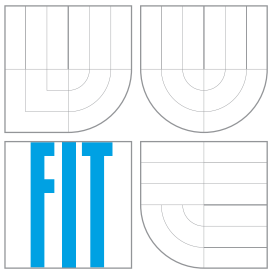
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN RAŠOVSKÝ

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

# MODULÁRNÍ PROSTŘEDÍ PRO ZPRACOVÁNÍ SENZORICKÝCH DAT

MODULAR FRAMEWORK FOR SENSOR DATA PROCESSING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MARTIN RAŠOVSKÝ
AUTHOR

VEDOUCÍ PRÁCE — Ing. KORČEK PAVOL
SUPERVISOR

BRNO 2015

# Abstrakt

Předkládáná bakalářská práce se zabývá problematikou návrhu a tvorby serverové aplikace pro systém inteligentní domácnosti, kde jejím zaměřením je poskytnout tomuto systému možnost výpočtů nad daty ve formě modulárního prostředí. Práce rozebírá a detailně popisuje architekturu systému, pro který je aplikace určena, se zaměřením na jeho serverovou část. Z této analýzy vychází fáze návrhu, která popisuje způsob komunikace aplikace s jejím okolím, navrhuje ukládání dat a konfiguraci. Dále práce provádí výběr vhodných technologií a rozebírá technické detaily realizace výsledné aplikace. Nakonec se práce věnuje způsobu testování a dokumentuje nasazení do určeného systému se zhodnocením funkčnosti celé aplikace, kde se také můžeme dočíst o možném dalším rozšíření práce.



# Abstract

The bachelor thesis deals with the issue of concept and realization of the server application for a smart home system, where the purpose of the whole application is in the form of framework that offers the calculations and possible actions among the system. The theoretical part of this thesis analyses and describes architecture of the existing system with the focus on the server part. From previous analysis is derived the concept of the application. The concept describes the way of communication with other nodes of the system, storing data and configuration. Further the thesis selects the used technologies and deals with the implementation details of the project. Furthermore the thesis documents testing, the integration into described smart home system and results of functionality of the application. In the conclusion is discussed further possible expansions of the project.



# Klíčová slova

inteligentní domácnost, serverová aplikace, modulární prostředí.

# Keywords

smart home, server application, framework.

# Citace

Martin Rašovský: Modulární prostředí pro zpracování senzorických dat, bakalářská práce, Brno, FIT VUT v Brně, 2015

# Modulární prostředí pro zpracování senzorických dat

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Korčeka

. . . . . .
Martin Rašovský
18. května 2015

## Poděkování

Rád bych touto cestou poděkoval panu PhD. Viktorovi Pušovi, za odborné vedení a podporu při tvorbě bakalářské práce a za mnoho podnětných informací týkajících se zvolené problematiky.

# Obsah

# Kapitola 1

# Úvod

Inteligentní dům či domácnost je objekt, který je automatizovaný a zajišťuje tak ideální prostředí pro jedince. Jediným faktorem ideálního prostředí nebývá jen komfort, ale také efektivita využití energetických zdrojů, schopnost autonomní správy objektu bez přítomných osob a v neposlední řadě také vzdálené sledování objektu a jeho správa pomocí koncových zařízení. Pojem inteligentní je zde mnohoznačný, protože zde zahrnuje všechny vlastnosti objektu definovaného výše. Jak jistý management nad objektem, monitorování objektu, ale i opravdu umělou inteligenci, která rozhoduje na základě jistých faktorů, zda se má či naopak nemá vykonat nějaká akce. Tyto faktory jsou obvykle zadávány uživatelem uvnitř objektu či kdekoli jinde, kde má uživatel přístup ke globální síti Internet.

Chceme-li tedy vytvořit systém inteligentní domácnosti, máme na výběr ze dvou možností. Budeme implementovat server s daty a veřejnou IP adresou s jednoduchým prostředím pro rozhodování správy jedné domácnosti nebo implementujeme výkonný server umístěný na páteřní lince s veřejnou IP adresou pro všechny tyto domácnosti, kde ovšem bude muset být sofistikované prostředí pro správu rozhodování úloh všech domácností.

Předmětem této bakalářské práce je návrh a vývoj prostředí do druhé varianty popsaného systému. Konkrétně je zadáním navrhnout a implementovat do již existujícího systému inteligentní domácnosti prostředí, které by mělo umožňovat vkládání samostatných aplikačních modulů, které budou provádět výpočty nad daty získanými v rámci inteligentní domácnosti a dle výsledků rozhodovat o provádění různých akcí. Z důvodu, že prostředí slouží zejména pro vkládání a obsluhu těchto samostatných modulů, tak budeme toto prostředí nazývat jako modulární prostředí (pro některé čtenáře známější pod pojmem framework). Data, nad kterými se budou tyto výpočty provádět budou získaná z teplotních, spínacích a jiných čidel v každé domácnosti (meteorologické veličiny jako teplota nebo i jiné, například záznam o sepnutí pohybového čidla). Navíc tyto aplikační moduly musí být možno zavádět, spouštět a pozastavovat při běhu tohoto prostředí.

V následující kapitole č. 2 si rozebereme architekturu již konkrétního systému inteligentní domácnosti, do které se bude modulární prostředí navrhovat a následně i implementovat. V další kapitole č. 3 provedeme vhodný návrh tohoto řešení s pomocí analýzy systému z předchozí kapitoly. Další kapitola č. 4 popisuje implementaci řešení, kapitola č. 5 kontroluje jeho funkčnost a hodnotí dosažené výsledky a v poslední kapitole č. 6 diskutujeme další možné rozšíření implementace.

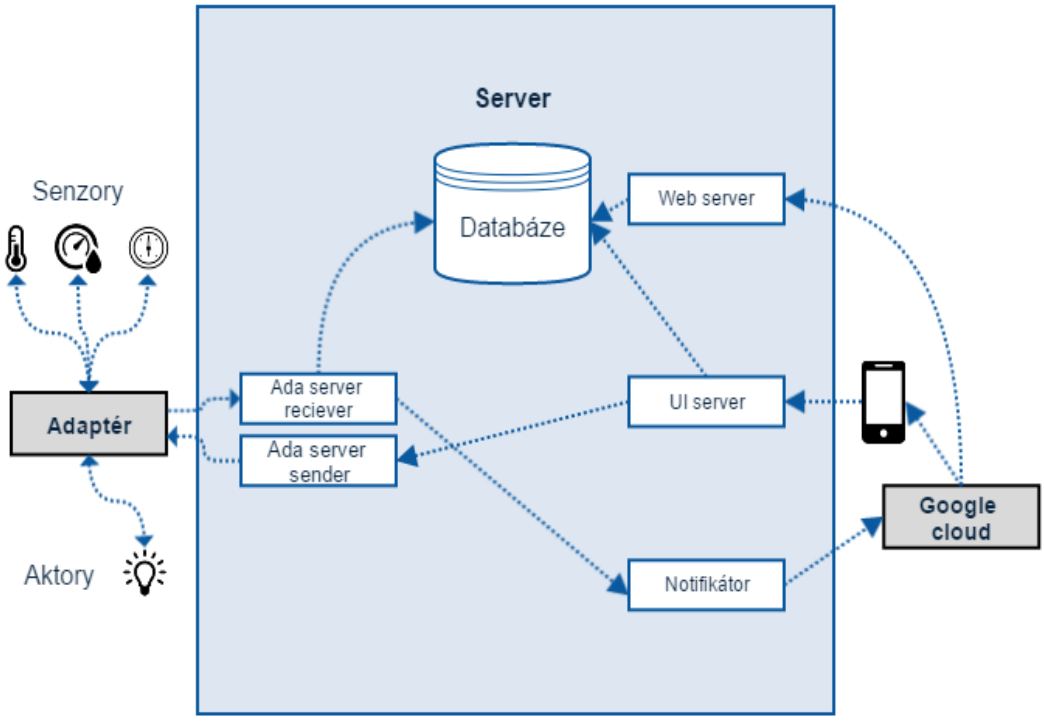
# Kapitola 2

# Popis existujícího systému

Kapitola popisuje stav vyvíjeného systému inteligentní domácnosti k datu 1. dubna 2015. Protože je systém pod neustálým vývojem, mohou se některé detaily po dobu psaní práce lišit. Nemělo by se však jednat o zásadní změny architektury, a ty, které jsou předmětem změn, jsou v textu explicitně uvedeny.

## 2.1 Architektura

Obecné schéma architektury systému inteligentní domácnosti je znázorněno na obrázku 2.1 níže. Architekturu systému můžeme rozdělit do tří ucelených částí. **Senzory (aktory) a jejich adaptér**, **server** a **mobilní zařízení**.




Obrázek 2.1: Diagram architektury inteligentního systému bez *modulárního prostředí*.

Senzory resp. aktory jsou elementární prvky celého systému, získávají různé veličiny o ovzduší resp. mohou měnit stav žárovky či vykonávat jiné akce. V blízkosti těchto senzorů je umístěn adaptér, který má na starosti správu těchto aktorů, komunikaci s nimi a získávání nových veličin. Zároveň má adaptér na starosti komunikaci s další částí systému, serverem. Server je další ucelenou částí architektury systému a má za úkol provoz různých služeb jako je *databáze*, uchovávající hodnoty jako např. veličiny přijaté ze senzorů, *Adapter server* komunikující přímo s fyzickými adaptéry a v neposlední řadě *UI Server*, který komunikuje přímo s mobilními zařízeními (viz. dále). Tyto služby jsou zde podrobněji naznačeny jako entity s orientovanými vazbami mezi sebou znázorňující komunikaci v příslušném směru. Jedna šipka znázorňuje jednosměrnou komunikaci, dvě šipky znamenají obousměrnou komunikaci. Jako poslední ucelenou částí je mobilní zařízení, které reprezentuje uživatelské rozhraní celého systému. Umožňuje asociovat senzory ke konkrétním adaptérům, zobrazovat aktuální stavy senzorů, vykreslovat grafy pomocí naměřených historických hodnot, vzdálenou správu aktorů apod. S mobilními zařízeními se komunikuje jak pomocí služby *UI server* na straně serveru, tak je zde využita i služba GCM [1] pro efektivnější zasílání informačních zpráv.

V následujících podkapitolách si rozebereme jednotlivé entity systému inteligentní domácnosti s jejich komunikacemi mezi sebou s důrazem na serverovou část a její služby, které se budou následně využívat při návrhu *modulárního prostředí*.

## 2.2 Senzory (aktory)

Senzory (i aktory) jsou bezdrátové koncové prvky, které se pomocí bezdrátového protokolu MiWi (od společnosti Microchip) připojují (párují) s adaptérem. Tyto prvky jsou navrženy pro co nejmenší spotřebu elektrické energie, aby mohly býti napájeny pomocí výměnných baterií. Pokud jsou tyto prvky využity jako časové spínače jiných zařízení, je možné je napájet zároveň ze sítě a dosáhnout tak vyšší spolehlivosti. Senzory dle svého typu snímají různé veličiny (senzor pohybu, senzory meteorologických veličin) a aktory zpravidla fungují jako spínače jiných zařízení jako např. automatické řízení venkovního osvětlení, řízení centrálního topení či klimatizace. Každý prvek má své jednoznačné označení, se kterým se přihlašuje k adaptéru. Veličiny jsou snímány periodicky po čase zadaném uživatelem. Nastavením větší periody může uživatel dosáhnout nižší spotřebu těchto zařízení.

## 2.3 Adaptér

Adaptér je zařízení připojené do vnitřní sítě, které si procesem párování registruje všechny senzory (aktory). Tímto párováním se provede jednoznačná identifikace koncových prvků, se kterými adaptér následně komunikuje. Párování senzorů s adaptérem je nutné, protože jak bylo výše zmíněno, adaptér nemůže odhadnout přesně dobu příchodu dat od koncových prvků.

Stejně jako senzory (aktory), tak i adaptér je jednoznačně identifikován pomocí jednoznačného identifikátoru, který je reprezentován pomocí QR kódu na každém adaptéru. Pro případ chyby čtení QR kódu je navíc přímo uveden v číselné podobě na zařízení.

Adaptér posílá přijaté zprávy na server pomocí protokolu TCP. Je zde řešen i případ, kdy nastane výpadek internetového připojení. Proto adaptér je vybaven SD kartou, kam se ukládají záznamy o datech, které nebyly odeslány na server.

Z důvodu vzdálené správy adaptéru je taktéž nutné, aby bylo možno adaptér aktualizovat. Navíc se předpokládá, že v budoucnu se budou využívat koncové prvky třetích stran. Proto na adaptéru

[1] Google Cloud Messaging for Android - https://developer.android.com/google/gcm/index.html

jsou spuštěny dvě aplikace. Aplikace pro správu adaptéru, který se neustále dotazuje, zda není něco potřeba a slouží např. pro aktualizaci firmwaru. Druhá aplikace se již stará o synchronizaci času adaptéru a ustavení zabezpečeného spojení se serverem, se kterým si následně vyměňuje získaná data.

## 2.4 Server

Server je výpočetně výkonné zařízení, které je připojeno k veřejné IP adrese s vysokorychlostním připojením ke globální síti Internet. Je to mezivrstva mezi směrovačem vnitřní sítě domácnosti či adaptérem a ovládacími zařízeními (mobilními telefony). Na této mezivrstvě je spuštěno několik služeb, které se starají o komunikaci s adaptéry, tak i s mobilními zařízeními. Zároveň je zde velice důležité úložiště dat (databáze). V následujících podsekcích si podrobně popíšeme jednotlivé služby spuštěné na tomto serveru, což je velice nutné pro pochopení následujícího návrhu řešení implementace *modulárního prostředí*.

### Ada server

Ada server je služba mající na starost přímou komunikaci s fyzickým adaptérem. Server pasivně čeká na data přijatá od adaptéru získaných ze senzorů a aktivně se stará o posílání příkazů na adaptér (ten je může dále odeslat na senzory či aktory). Může se např. jednat o nastavení výstupního proudu aktoru nebo změnu periodicity senzoru.

Z tohoto popisu vyplývá, že je tato služba rozdělena na dvě sémanticky oddělené části. Na obrázku 2.1 vidíme, že je tato služba rozdělena na dvě aplikace, kde každá z nich má vlastní konfigurační soubor.

*Ada server reciever* je aplikace, která přijímá data z adaptérů, ukládá je do databáze a zároveň jim jako odpověď posílá čas do nového buzení senzoru. Pracuje jako konkurentní server a její úlohou je čekání na nový požadavek o spojení směrem od adaptéru. Vytvoří jedno vlákno při připojení adaptéru, kterým obslouží pouze jednu zprávu přijatou od adaptéru a potom sama spojení ukončí a čeká na znovu připojení adaptéru (typicky když adaptér znovu získá nová data od senzorů). Při jeho navázání spojení na server služba zkontroluje, zda se zde tento adaptér již s tímto identifikátorem nachází v databázi. Pokud se nachází, pak pouze aktualizuje daný záznam s adaptérem. Pokud nikoliv, tak se uloží jako záznam nový. Taktéž se provede přidání do databáze všech aktorů (senzorů). Následně se odešle odpověď adaptéru, kdy se mají koncové prvky znovu probudit (reálně implementováno před zjištěním, zda se adaptér již nachází v databázi). A nakonec se provede uložení naměřených hodnot ze senzorů do databáze, ovšem pouze pokud je toto ukládání dat pro příslušné koncové zařízení povoleno. Tomuto ukládání dat se v našem systému říká tzv. *logování*.

Jak je z obrázku 2.1 vidět, mezi *Ada server reciever* a *Notifikátorem* je naznačena vazba. Je to z toho důvodu, že zde není žádná služba, která by právě přijatá nová data vzala a provedla nad nimi výpočet. Proto na jednoduché problémy bylo prozatím vyhodnocení senzorických dat prováděno na *Ada server*u, kdy se např. při dosažení příliš velké teploty na senzoru odeslala zpráva uživateli na jeho mobilní ovládací zařízení. Formát konfiguračního souboru této části *ada server*u je naznačen v kódu 2.1.

```
<configuration>
    <Database>home4</Database>
    <Port>7080</Port>
    --Počet současně obsluhovaných spojení
```

```
    <ConnectionsCount>10</ConnectionsCount>
    --Doba čekání při neaktivním spojení
    <ConnectionTimeOut>10</ConnectionTimeOut>
    <LogConfig>
        <Level>7</Level>
        <FileNaming>ada_server_receiver</FileNaming>
        <FilesCount>5</FilesCount>
        <LinesCount>100</LinesCount>
        <LogPath>/home/tuso/log/</LogPath>
    </LogConfig>
</configuration>
```

Kód 2.1: Konfigurační soubor pro *Ada server reciever*

*Ada server sender* je serverová aplikace, která čeká na spojení od serverové části komunikující s uživatelem (*UI Server*) a vytváří tak spojení směrem k adaptérům. Konfigurace této aplikace se provádí pomocí konfiguračního souboru, jehož formát je podobný jako pro *Adapter Server Reciever*, viz kód 2.2.

```
<configuration>
    <Database>home4</Database>
    <Port>7081</Port>
    <ConnectionsCount>10</ConnectionsCount>
    <AdapterPort>7978</AdapterPort>
    <LogConfig>
        <Level>7</Level>
        <FileNaming>ada_server_sender</FileNaming>
        <FilesCount>5</FilesCount>
        <LinesCount>100</LinesCount>
        <LogPath>/home/tuso/log/</LogPath>
    </LogConfig>
</configuration>
```

Kód 2.2: Konfigurační soubor pro *Ada server Sender*

### Databáze

Pro ukládání veškerých dat celého systému je zde využit objektově relační databázový systém **PostgreSQL**. Schéma databáze je uvedeno na obrázku v příloze B.1.

### Notifikátor

Notificator je knihovna pomocí níž se mohou odesílat notifikace (zprávy) uživateli na koncové mobilní zařízení. Tato knihovna prozatím funguje pouze pomocí služby **Google Cloud Messaging**. Umožňuje odeslat zatím 4 různé typy zpráv: informační zprávy (*InfoNotification*), výstražné zprávy (*AlertNotification*), reklamní zprávy (*AdvertNotification*) a kontrolní zprávy (*ControlNotification*). Vývojář si vybere příslušnou požadovanou zprávu, vytvoří její instanci a předá jako parametr třídě *Notificator* se statickou metodou `sendNotification()`. Pro úplnost je zde uveden její UML třídní diagram v příloze B.2

### UI server

UI server je služba, která komunikuje s mobilním koncovým zařízením. Pracuje jako server aplikace a pasivně čeká na zprávu od mobilního zařízení reprezentující požadavek uživatele (např. změň periodu snímání veličin aktorem).

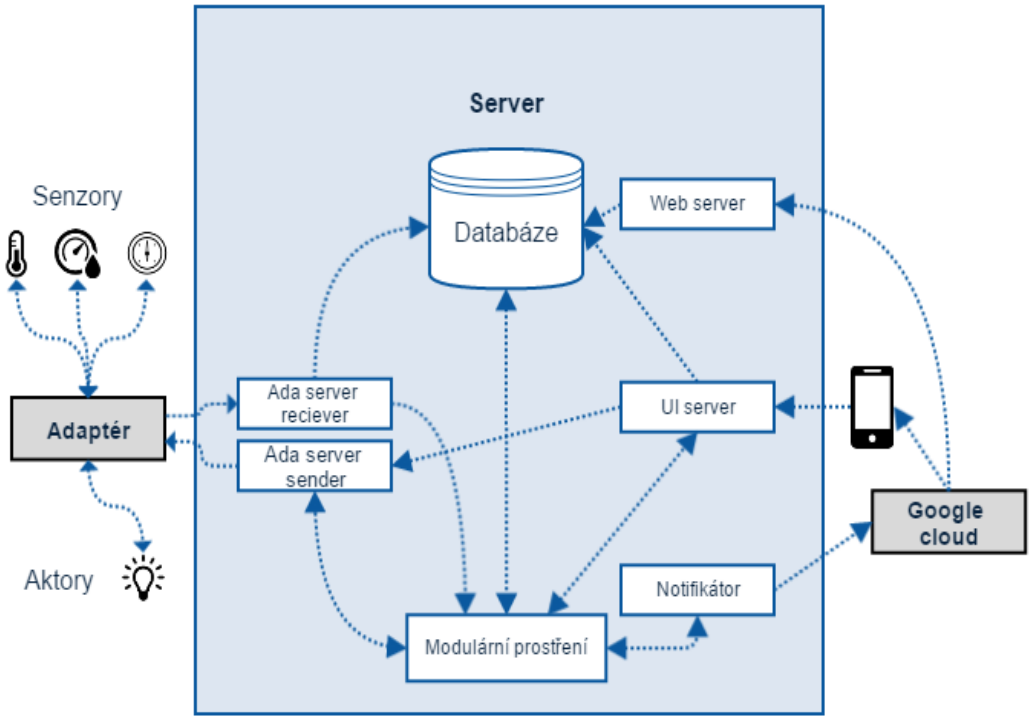
# Kapitola 3

# Návrh modulárního prostředí

Tato kapitola popisuje návrh *modulárního prostředí*, které je tvořeno přesně na míru do systému popsaného v minulé kapitole, protože neexistuje žádné jiné univerzální prostředí či framework, který by se dal namísto tohoto prostředí použít.

Nejprve si popíšeme jak je navrženo vložení *modulárního prostředí* z pohledu serveru a popíšeme si základní mezi-procesovou komunikaci. Jak již bylo naznačováno výše, bylo nutné v návrhu změnit architekturu stávajícího řešení uvedené v minulé kapitole. Změna se dotýká *Ada server*u, *UI server*u a databáze (bude vysvětleno níže).

*Modulární prostředí* bylo umístěno na server jako další autonomní služba komunikující s okolím. Na obrázku 3.1 oproti obrázku z předchozí kapitoly 2.1 je vidět, jak se systém změnil po vložení modulárního prostředí.




Obrázek 3.1: Diagram architektury inteligentního systému s *modulárním prostředím*.

Bylo zde vloženo *modulární prostředí* a byla zde zrušena vazba od *Ada server reciever* k *Notifikátoru*, který nyní již využívá naše prostředí.

Z obrázku 3.1 je vidět, že navržená služba komunikuje téměř se všemi ostatními službami na serveru. Bylo to nutné takto navrhnout zejména z toho důvodu, že modulární prostředí musí mít plnou podporu správy nad celým systémem. Tento požadavek vyplývá z nutnosti poskytnout autorům aplikačních modulů co nejvíce možných akcí, které se dají provádět nad celým systémem celé inteligentní domácnosti (jak přístup do databáze, tak manipulovat s reálnými aktory apod.).

*Modulární prostředí* proto umožňuje přijímat aktuální hodnoty ze senzorů (*Ada server reciever*), měnit stav aktorů (*Ada server sender*), umožňuje posílat informační zprávy (*Notifikátor*), dokáže zpracovávat požadavky odeslané uživatelem z mobilního zařízení a posílat na něj odpovědi (zprostředkovaně skrze *UI Server*) a nakonec má podporu práce s databází, kde také uchovává data, která musí být persistentní.

Jak to navrhnout takovým způsobem, aby toto vše bylo možné podporovat prostředím, si rozebereme v následující podkapitole.

## 3.1 Návrh vnitřní struktury prostředí

Celé prostředí je rozděleno do dvou navzájem samostatných částí. Do *modulárního prostředí (hlavního programu)* a *aplikačních modulů*.

**Modulární prostředí**, nyní již ve smyslu hlavního programu celé aplikace, má za hlavní úkol v přesně definovanou dobu a za daných okolností spustit aplikační modul. Je dále rozděleno na:

- komunikační část,
- část pracující nad databází,
- část starající se o možnost načítání aplikačních modulů za běhu aplikace.

**Aplikační modul** je program, který má na starosti výpočet nad daty získanými z modulárního prostředí, má vnitřní podporu různých funkcí pro ovlivňování celého systému inteligentní domácnosti prostřednictvím komunikace s *modulárním prostředím*.

Z výše uvedených popisů obou částí vyplývá, že elementární na celém systému je komunikace mezi těmito dvěma částmi. Při spuštění aplikačního modulu musí být z modulárního prostředí předány informace, které budou nezbytně nutné pro výpočet. Dle specifikace systému, do kterého se prostředí zpracovává, je nutno každému aplikačnímu modulu předat alespoň tyto informace:

- informace o uživateli, pro kterého je spuštěn,
- informace o adaptéru, pro který je modul spuštěn,
- informace o samotném aplikačním modulu (jeho identifikační číslo, typ . . . ),
- informace o databázi (její název).

Uvedené informace budou následně v aplikačním modulu použity pro komunikaci s *modulárním prostředím* a pro získávání dat přímo z databáze. Dále je možno aplikačnímu modulu předat data, nad kterými bude provádět výpočty. Protože data mohou nabývat různých hodnot a dat předaných do modulu může být různý, tak je pro autory *aplikačních modulů* navržen formát pro definování těchto předaných dat. Jde o seznam názvů dat a jejich typů (viz dále).

Aby hlavní aplikace *modulárního prostředí* předem věděla jaká data má při spouštění *aplikačnímu modulu* předat, je potřeba definovat *aplikační moduly* v jistém formátu. Pro autory *aplikačních modulů* je navržen tento formát:

- seznam názvů dat a jejich typů,
- název aplikačního modulu,
- jedinečný identifikátor *aplikačního modulu* uvnitř celého modulárního prostředí.

Je nutné zdůraznit, že se nejedná o žádný formát, který bude přebírat výsledná aplikace. Jedná se o obecnou definici *aplikačního modulu*. Tyto obecné definice aplikačních modulů jsou určeny k tomu, aby zbytek systému inteligentní domácnosti byl informován o tom, jaké přesně parametry se budou do aplikačního modulu předávat, v jakém přesném pořadí a jakého přesného typu budou (bez následné nutné znalosti celého *Modulárního prostředí*). V tabulce 3.1 je uvedena šablona pro tuto definici *aplikačního modulu* a následně další tabulka 3.2 popisuje možné typy dat pro definice parametrů v rámci definice jednoho *aplikačního modulu*. Je nutno zdůraznit fakt, že při špatném následném dodržení definice *aplikačního modulu* je navrženo *modulární prostředí* takovým způsobem, že požadavek o výpočet takového špatně zadaného modulu zahodí. Aktuální seznam definic i včetně aktuálního popisu typů hodnot je veden na webových stránkách projektu inteligentní domácnosti [1].

| | |
|---|---|
| NÁZEV APLIKAČNÍHO MODULU | ID |
| | |
| NÁZEV PARAMETRU | TYP |
| NÁZEV PARAMETRU | TYP |
| NÁZEV PARAMETRU | TYP |

Tabulka 3.1: Šablona pro definici *aplikačního modulu*.

| **typ** | **popis** |
|---|---|
| `ANY_SENZOR_VALUE` | Jakákoli hodnota přijatá ze senzoru. Uživatel si konkrétní typ hodnoty zvolí a algoritmu bude předán. |
| `ANY_ACTOR_VALUE` | Identifikace aktoru, který se má změnit z jedné hodnoty na druhou. |
| `KONKRÉTNÍ DATOVÝ TYP` | (`INT`, `REAL` , `DOUBLE`, `STRING`, . . . ) |
| `ENTITY` | (gt, lt) Znak jako HTML entita. gt pak reprezentuje znak > |
| `DIRECTION` | (in, out) Určuje směr překročení hranice geolokační oblasti |
| `NOTIF_OR_ ACTOR` | (notif, act) Určuje zda následující parametr po tomto parametru bude text notifikace či identifikace aktoru. |

Tabulka 3.2: Typy dat předávaných v parametrech dle definice *aplikačního modulu*.

Na definovaném rozhraní mezi těmito dvěma částmi je závislý celý zbytek návrhu. V následujících podsekcích popíšeme odděleně tyto dvě části a jakým způsobem jej toto rozhraní ovlivnilo.

[1] Webová stránka projektu se seznamem definovaných aplikačních modulů. https://ant-2.fit.vutbr.cz/projects/server/wiki/Alg_specification_list

### 3.1.1 Hlavní aplikace modulárního prostředí

Na obrázku 3.2 vidíme diagram hlavní aplikace se všemi jejími podčástmi. Nejprve si rozebereme každou jednotlivou část hlavní aplikace *modulárního prostředí*, následně si mezi nimi vysvětlíme vazby a zároveň si popíšeme za jakých okolností se mají spouštět již definované aplikační moduly.

#### Konfigurátor

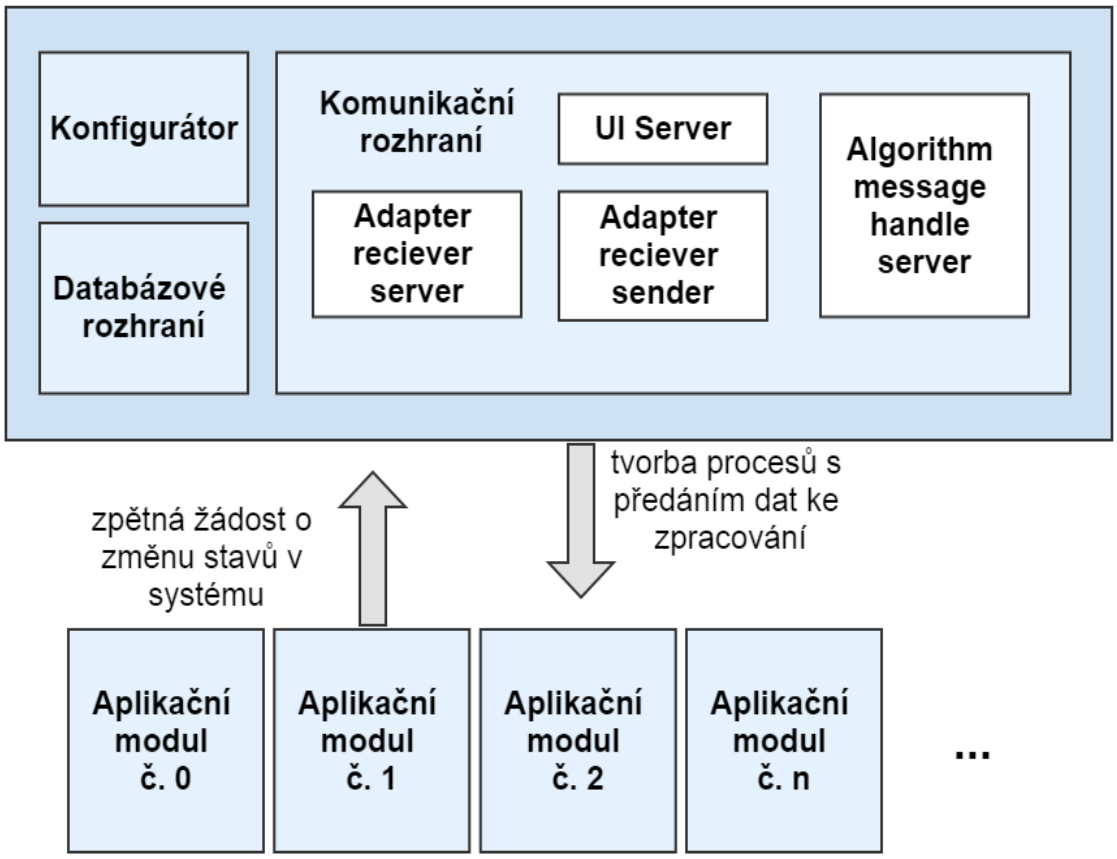
*Konfigurátor* je část aplikace *modulárního prostředí*, která se stará o načtení konfiguračního souboru, ve kterém lze nastavit *modulární prostředí* bez nutného zásahu do kódu aplikace. Jsou zde uloženy následující informace:

- porty, na kterých se bude komunikovat,
- název databáze,
- maximum možných databázových spojení,
- velikost vyrovnávací paměti ve slabikách při příjmu dat (během komunikace),
- cesty ke konfiguračnímu souboru se seznamem definic aplikačních modulů (viz. 3.1.2),
- nastaveni tzv. logování do souboru.

Logování v tomto textu znamená zaznamenávání jistých zpráv *modulárním prostředím* do souboru, pomocí nichž se může zpětně při havárii či ladění programu dohledat problému. Tomuto logování se tedy v rámci konfiguračního souboru dá nastavit, jaký má být prefix názvu logovacího souboru, název aplikace, jaká logování provádí (kvůli přehlednosti a znovupoužitelnosti), míra logování (je zde více typů možných zpráv do logovacího souboru a míra logování určuje, zda má vypisovat i ty výpisy, které např. neznamenají výpis o chybě), dále se zde dá nastavit maximální počet logovacích souborů, které aplikace vytvoří a počet záznamů na jeden soubor. V kódu 3.1 můžeme vidět jeho přesný formát.

```
<framework_config
      portUIServer="7082"
      portAdaRecieverServer="7083"
      portAdaSenderServer="7081"
      portAlgorithmsServer="7084"
      DBName="home6"
      maxNumDBConnections="30"
      recieveBuffSize="1000"
      algorithmsConfig="Algorithms/AlgsConfig.xml">
      <loger
            fileName="log"
            appName="framework"
            verbosity="7"
            filesCnt="5"
            LinesCnt="100"
      />
</framework_config>
```

Kód 3.1: Konfigurační soubor *modulárního prostředí*.

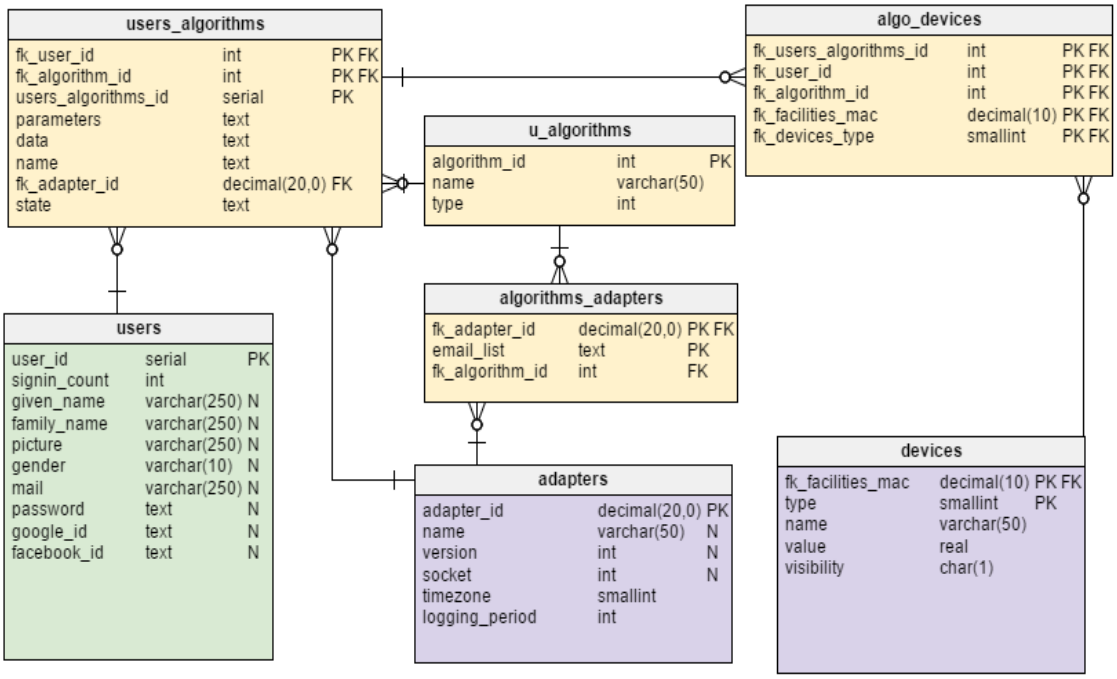



Obrázek 3.2: Diagram návrhu částí hlavní aplikace *modulárního prostředí*.

### Databázové rozhraní

Databázové rozhraní reprezentuje sadu funkcí pro práci s databází. Je to nutné, protože hlavní aplikace zde ukládá informace, pro které je nezbytné, aby byly persistentní. Návrh byl nejprve prováděn definováním požadavků a následném navržením jeho schématu. Nové tabulky přidané do schématu vidíme na obrázku 3.3

Vidíme zde, že ve schématu přibyly z důvodu *modulárního prostředí* 4 tabulky. Tabulka **u_algorithms** definuje názvy jednotlivých *aplikačních modulů* a jejich jedinečný identifikátor a typ (přidáno do budoucna kdy se počítá více typů různých aplikačních modulů). Tabulka **users_algorithms** definuje jaký algoritmus s jakými konkrétními hodnotami si uživatel nadefinoval. Doména *parameters* reprezentuje zadané konkrétní hodnoty, jejichž typ (sémantika) byl zadán v definici *aplikačního modulu*. Doména *data* je zde pro ukládání případných hodnot pro autory aplikačních modulů. *Name* umožňuje uživateli pojmenovat vlastní modul. Doména *state* je zde pro splnění požadavků této práce a znamená, zda je *aplikační modul* ve stavu zapnuto či vypnuto. Uživatel si tedy již jednou nadefinovaný *aplikační modul* může kdykoli znovu spustit. Vzhledem k tomu, že jde o aplikační moduly zpracovávající senzorická data, tak je na každý *aplikační modul* navázáno 0 až X senzorů či aktorů. Seznam senzorů resp. aktorů pro každý nadefinovaný uživatelský modul je uložen v tabulce **algo_devices** s potřebnými informacemi o konkrétním zařízení.




Obrázek 3.3: Databázové schéma s tabulkami pro *modulární prostředí*.

### Komunikační rozhraní

Komunikační rozhraní je rozhraní, které komunikuje jak s okolními službami běžícími na straně serveru, tak přijímá zprávy od spuštěných aplikačních modulů. Je zodpovědné nejen za správné přijetí či odeslání dat, ale i za správnou odpověď zpět na druhou stranu komunikace, pokud to situace vyžaduje (např. zaslání informace o chybových kódech). Veškerá komunikace využívá zpráv ve formátu XML. Ohledně jazyka XML se můžete více dočíst například v publikaci (Kosek, 2000).

Komunikace s *Adapter reciever server* spočívá v příjmu dat naměřených ze senzorů, které jsou přeposílány do *modulárního prostředí*. Příklad přijaté zprávy z tohoto serveru je uveden v kódu 3.2.

```
<adapter_server adapter_id="0x00face" fw_version="2.02"
protocol_version="1.0" state="data" time="1430253897">
    <device id="0x001fac">
        <battery value="1456" />
        <rssi value="90" />
        <values count="2">
            <value offset="0x00" type="0x0a">
                23.00
            </value>
            <value offset="0x00" type="0x01">
                74.00
            </value>
        </values>
    </device>
```

```
</adapter_server>
```

Kód 3.2: Příklad přijaté XML zprávy od *Adapter reciever server*.

Komunikace s *Adapter server sender* přináší podporu pro nastavování fyzických senzorů resp. aktorů. Jelikož *adapter sender* je server, a na každý dotaz zasílá odpověď, tak tato komunikační část musí také mít na starosti správu případných problémů při přijetí chybových zpráv z druhé strany. Zprávy definované pro komunikaci s fyzickými senzory či aktory jsou definovány na webových stránkách projektu inteligentní domácnosti [2]. V kódu 3.3 je uveden příklad zprávy o přepnutí aktoru na stav zapnuto.

```
<request type="switch">
    <sensor id="1.1.1.1" type="0x00" onAdapter=12345">
        <value>ON</value>
    </sensor>
</request>
```

Kód 3.3: Příklad XML zprávy zasílané na *Adapter server sender* pro přepnutí aktoru do stavu zapnuto.

Komunikace s *UI server*em zpřístupňuje komunikaci s uživatelem skrze mobilní zařízení. Například uživatel skrze tuto komunikaci zasílá do *modulárního prostředí* požadavek o přidání *aplikačního modulu* s jeho nadefinovanými parametry. Tato komunikace je navržena dle modelu klient-server (Matoušek, 2014), kdy je komunikační UI část serverem a čeká na požadavky od uživatele (klienta). Na každý požadavek je definována přesná odpověď a jsou tak předem stanovena přesná pravidla pro komunikaci. To definuje komunikační protokol, který je popsán na webových stránkách projektu inteligentní domácnosti [3]. Pro úplnost je zde v kódu 3.4 uvedena zpráva z telefonu na tuto komunikační část, která má přidávat aplikační modul do *modulárního prostředí*. Jakým způsobem se to provede, bude popsáno dále.

```
<com
    ver="x.x"
    bt="42"
    state="addalg"
    algname="uzivatelem definovane jmeno"
    aid="64206"
    atype="3" >
        <dev id="12345" type="1" pos="1"/>
        <par pos="1">42</par>
        <par pos="2">Hoří mi dům</par>
        <geo rid="478547" type="in" />
</com>
```

Kód 3.4: Příklad XML zprávy zaslané z mobilního zařízení a přeposlané *UI server*em do *modulárního prostředí*.

### Spolupráce částí hlavní aplikace

Výsledná hlavní aplikace se sestává ze všech dříve popsaných částí, které spolupracují a vytváří tak ucelený celek. Spouštění samostatných *aplikačních modulů* je vyvoláváno ve dvou samostatných

[2] https://ant-2.fit.vutbr.cz/projects/server/wiki/Protokol_ui_server-ada_server.
[3] https://ant-2.fit.vutbr.cz/projects/android-app/wiki/Smarthome_Phone-Server_Protocol_XML_

případech.

První případ je definován ze zadání. *Aplikační modul* je vyvolán, když do *modulárního prostředí* skrze komunikační rozhraní jsou přijata data ohledně senzoru, pro který si uživatel nadefinoval tento *aplikační modul*. Pak je pro tohoto uživatele spuštěn konkrétní *aplikační modul* a jsou jemu předány parametry uložené v databázi. Navíc jsou mu předány nezbytně nutné informace jako pro každý *aplikační modul*.

Druhý případ je rozšířením tohoto *modulárního prostředí*, protože umožňuje navíc vyvolat spuštění *aplikačního modulu* z důvodu požadavku zaslaného přímo z mobilního zařízení. Je pro něj navržen speciální typ *aplikačního modulu*, takže se vždy při příchodu této zprávy od uživatele vyhledají v databázi všechny záznamy o tomto algoritmu definovaném tímto uživatelem a těmto *aplikačním modulům* jsou stejně jako v předchozím případě předány parametry a nezbytně nutné informace o *aplikačním modulu* a o uživateli, který nadefinoval tento *aplikační modul*.

### 3.1.2 Návrh aplikačního modulu

*Aplikační modul* je program, pro který jsou předem definována data, která očekává. Z toho důvodu je pro *modulární prostředí* navržen konfigurační soubor ve formátu XML, ve kterém budou uloženy veškeré informace o aplikačních modulech. Kód 3.5 popisuje jeho navržený formát. Format vyplývá již z výše uvedených definic algoritmů. Nyní si rozebereme tento XML konfigurační soubor podrobněji.

```
<algorithms_config>
    <alg
        id="1"
        name="watch_and_notify"
        pathOfBinary="Algorithms/watch_and_notify"
        numParams="5"
        maxDevs="2"
        type="1"
    />
    <alg
        id="3"
        name="geofencing"
        pathOfBinary="Algorithms/geofencing"
        numParams="5"
        maxDevs="1"
        type="2"
    />
</algorithms_config>
```

Kód 3.5: Formát konfiguračního souboru pro *aplikační moduly*.

První atribut (`id`) je jedinečný identifikátor aplikačního modulu. Atribut `name` znamená název *aplikačního modulu* a zároveň název binárního souboru ke spuštění tohoto *aplikačního modulu*. Atribut `pathOfBinary` obsahuje relativní cestu k binárnímu spustitelnému souboru *aplikačního modulu*. Atribut `numParams` znamená počet záznamů v definici algoritmu, neboli počet dat zaslaných jako uživatelské parametry do *aplikačního modulu*. Atribut `maxDevs` je maximální počet zařízení (senzorů). Poslední atribut `type` rozlišuje zda se jedná o *aplikační modul*, který bude vyvolávaný příchodem dat ze senzorů (`type`="1") či zda se bude *aplikační modul* vyvolávat při příchodu dat z mobilních zařízení (`type`="2").

Nyní máme navržen aplikační modul, který získává data v určitém pořadí a formátu. Autor tedy dle těchto přesných definic ví, kde data získal a může nad nimi provádět výpočty a na základě nich provádět akce. Tyto akce provádí prostřednictvím *modulárního prostředí*, takže po dokončení operací (výpočtů) nad daty musí zase komunikovat se serverem, kterému předá informace o tom, jaké akce má *modulární prostředí* vykonat. V tomto návrhu je pouze jediná vyjímka, a to ve způsobu práce nad databází. Aby se omezilo na co nejmenší komunikaci s *modulárním prostředím*, tak jako jedinou nezprostředkovanou akci nad systémem je komunikace s databází. *Aplikační modul* tak získá možnost ukládat data pro budoucí použití, pro které je v databázi v tabulce `users_algorithms` navržena doména `data`, která je přímo určena k tomuto ukládání dat pro budoucí použití. Navíc aplikační modul bude moci získat veškeré jiné informace nad systémem, které bude potřebovat. Není možné omezovat sílu *aplikačního modulu*, protože pro budoucí *aplikační moduly* nemůžeme vědět, jaké výpočty se zde budou provádět.

Tímto je navrženo *modulární prostředí* se všemi klíčovými požadavky, aby mohlo správně pracovat a splňovalo zadání práce. V následující kapitole si již rozebereme jakým způsobem je toto prostředí implementováno a jak jsou problémy řešeny již na konkrétní úrovni.

# Kapitola 4

# Implementace dle zvoleného návrhu

Hlavním a jediným požadavkem pro řešení implementace tohoto *modulárního prostředí* bylo zvolit takový programovací jazyk, který je vhodný pro linuxové prostředí.

Pro tuto práci byl zvolen programovací jazyk C++, který vyvinul Bjarne Stroustrup a je rozšířením jazyka C. Důvodem volby tohoto programovacího jazyka bylo, že podporuje objektově orientované paradigma, poskytuje knihovny s podporou BSD schránek pro síťovou komunikaci, a také knihovny pro tvorbu vláken (sdílení paměti) a jeho zdrojové soubory se překládají do spustitelného programu (strojový kód), který je při překladu optimalizován a následně při běhu programu není závislý na žádném interpretu či virtuálním stroji (jako by to bylo např. u jazyku Java). Navíc linuxové prostředí poskytuje pro tento jazyk řadu nástrojů pro editaci jeho zdrojových kódů, nástroje pro překlad (GCC [1]), a také nástroje pro ladění a profilování programů (např. DDD [2] či Valgrind). V poslední řadě je třeba volbu jazyka odůvodnit tím, že zbytek systému inteligentní domácnosti běžící na serverové části byl již v tomto jazyce implementován (*Ada Server*, *UI Server*) a proto bylo vhodné tento jazyk dodržet pro následné možné sdílení kódu (znovupoužitelnost).

## 4.1 Implementace hlavního programu Modulárního prostředí

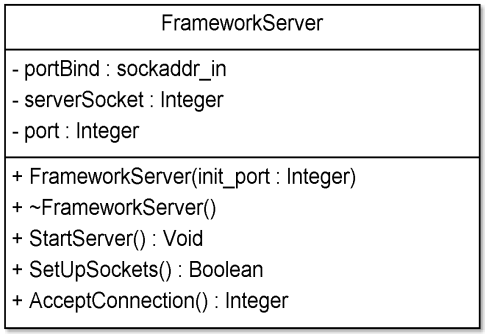
Nyní si rozebereme implementaci hlavní části aplikace, jak je program prováděn od jeho spuštění až po jeho ukončení. Pro chronologičnost textu si nejdříve popíšeme činnost hlavní aplikace bez technických detailů o práci s databází, jakým způsobem jsou spouštěny aplikační moduly, jak konfigurátor nastavuje *modulární prostředí* při jeho spuštění a jakým způsobem pracuje logování informací do souboru. To z toho důvodu, že tyto části jsou abstrahovány takovým způsobem, že je možné jim bez potíží zcela porozumět z názvů metod a čtenář se nejprve dozví nejpodstatnější informace o řešení tohoto systému.

Hlavní část aplikace *modulárního prostředí* je implementována jako skupina tří konkurentních serverů sdílejících paměť pomocí vícevláknového programování. Jde o server pro zpracování zpráv od *Adapter Server Reciever*, server pro zpracování zpráv od *UI Server*u a třetí server se stará o zpracování zpráv přijatých jako požadavky od *aplikačních modulů*.

Pro všechny tři konkurentní servery je implementována třída `FrameworkServer` viz kód 4.1, pomocí níž program vytvoří 3 objekty a následně dle nastavení v konfiguračním souboru předá port v konstruktoru do atributu třídy `port` a dle tohoto portu se pak dále rozlišuje jakým způsobem má server zprávy obsluhovat (příslušné porty se předem nastavují v konfiguračním souboru, viz 3.1.

[1] GNU Compiler Collection - https://gcc.gnu.org/

[2] The Data Display Debugger - http://www.gnu.org/software/ddd/




Obrázek 4.1: Třída FrameworkServer.

Následně jsou tyto objekty předány pro tvorbu vláken, kdy je vyvolána metoda `StartServer`, která volá dvě metody `SetUpSockets()` a `AcceptConnection()`. Metoda `SetUpSockets()` vytvoří BSD schránku a nastaví pomocí opreace bind(), kterou podporuje knihovna BSD socket, příslušný port. Dále nastaví naslouchání na tomto portu (operace `listen()`). Metoda `AcceptConnection()` je hlavní částí konkurentního serveru. Zde dochází k čekání na příchozí spojení (pomocí operace `accept()`) od klienta a při příchozím spojení je vytvořen objekt ze třídy `FrameworkServerHandle`, viz obrázek 4.2.

Tomuto objektu je předán port serveru, na kterém právě naslouchá (pro budoucí rozpoznání jak má na zprávu reagovat) a dále předána schránka s nově vytvořeným spojením.

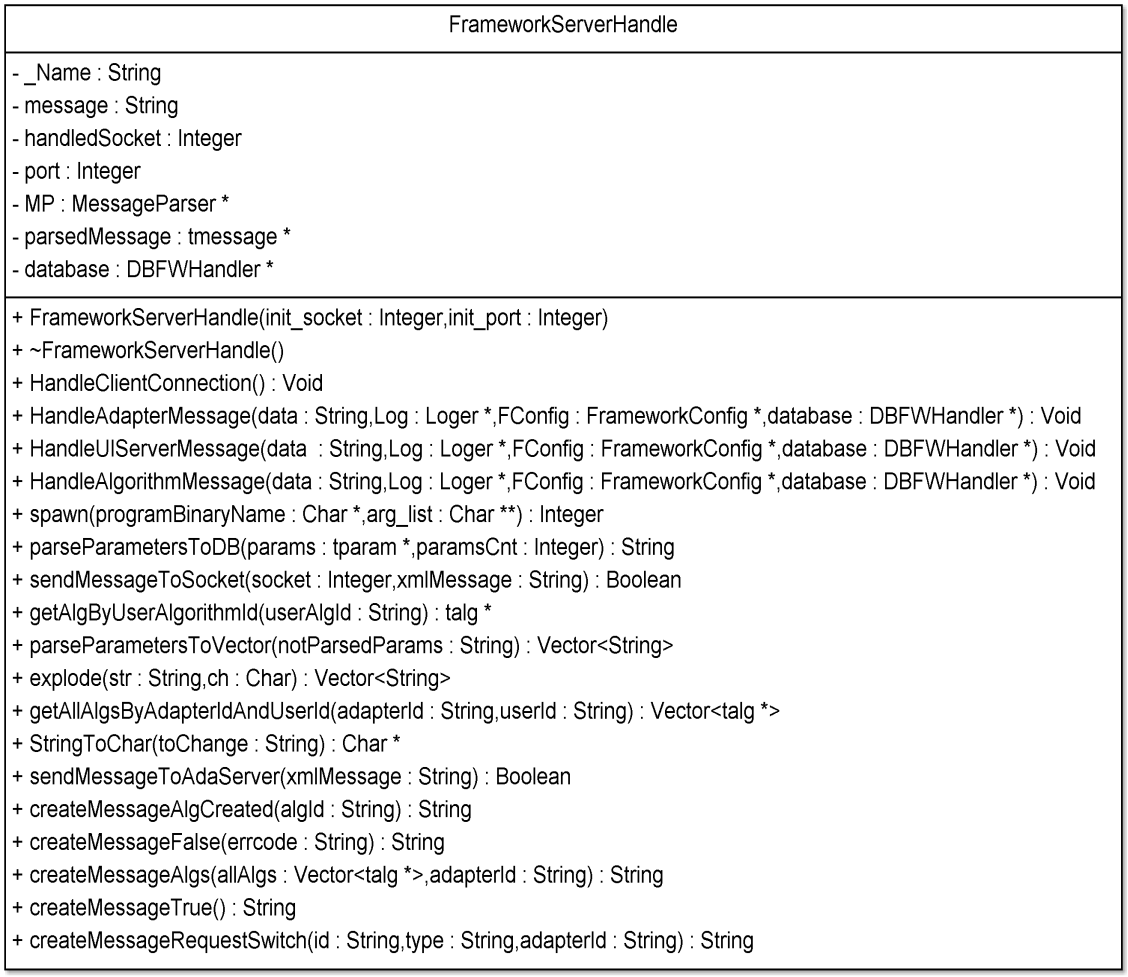
Podobně jako při tvorbě serverů je i pro obsluhu zprávy na každém z těchto serverů vytvořeno vlákno, kterému je předán tento objekt s nastavenou počáteční metodou `Handle Client Connection()`. Navíc je zde řešeno odchycení vyjímky pokud nastanou problémy s alokací paměti vlákna, aby celá aplikace následně dále reagovala na další spojení.

Z definice konkurentního serveru vyplývá, že běží, dokud mu není předán požadavek o ukončení. Ukončení těchto tří konkurentních serverů (i zbytku celé aplikace) s řádným uvolněním paměti je umožněno pomocí zaslání signálu `SIGTERM` aplikaci (viz. podsekce 4.1.7).

Instance třídy (`FrameworkServerHandle` 4.2) má zodpovědnost za zpracování zprávy, ale nestará se o to, jakým způsobem jí bylo předáno spojení a zda jí byl předán správný port.

V tomto novém vlákně se z předaného objektu vyvolá metoda `HandleClientConnection`, která již pracuje s přijatým spojením. Nejprve v této metodě dochází pomocí předaného portu v objektu k rozlišení, z jakého serveru byla tato obsluha zprávy vyvolána (dle toho se nastaví pravdivostní hodnota do příslušné proměnné). Pokud by port neodpovídal ani jednomu ze tří portů, pak by došlo k nevykonání ani jedné ze tří typů obsluhy.

Všechny tři servery mají v obsluze zprávy společné získání jednoho databázového spojení do atributu objektu `database` pomocí kontejneru databázových spojení `DBConnections Container` (viz. podsekce 4.1.4), které se bude používat v daném vláknu (obsluze spojení) a dochází k samotnému čtení zprávy. Čtení zprávy je implementováno pomocí operace `poll()`, která pracuje tím způsobem, že je jí nastaven socket, na kterém čeká na příjem dat. Je možné jí také nastavit jistý čas, po který pokud nepřijdou žádná data, tak dojde k vyvolání chyby výpadku spojení (tzv. timeout). Zde je timeout nastaven na hodnotu 1000 ms. Pomocí této funkce `poll()` je zde implementována tedy možnost reagovat na timeout spojení a dále možnost, kdy nastala jiná chyba při čtení dat (návratová hodnota z funkce `poll()` je rovna -1). Pokud nenastane problém a přijdou data korektně (dle návratové hodnoty z funkce `poll()`), pak dle typu obsluhy je očekávaný jistý formát zprávy. Zprávy jsou dle návrhu řetězce ve tvaru XML, tedy očekávaný konec zprávy je vždy koncový z párového kořenového tagu zprávy. Pro *UI Server* zprávy je to `</com>`, pro Adapter server zprávy




Obrázek 4.2: Třída FrameworkServerHandle.

`</adapter_server>` a pro zprávy od algoritmů je to řetězec `</alg_m>`.

Zpráva se následně ve všech třech případech uloží do proměnné `data` (`std::string` a následně je zpracovaná pomocí externí knihovny pugixml (Kapoulkine, 2006) a převedena do objektu, ze kterého se již dají číst data této zprávy. Tato knihovna pugixml je zde využita z toho důvodu, protože není předmětem této práce psát parser XML zpráv a toto je právě knihovna k tomuto účelu určená s otevřenou licencí. I tak není práce s touto knihovnou příliš elegantní.

V dalším kroku metoda `HandleClientConnection()` dle typu obsluhy zprávy vyvolá jednu ze tří metod - `HandleAdapterMessage`, `HandleUIServer Message` a `Handle Algorithm Message`, kde každá z nich implementuje již konkrétní chování obsluhy zprávy zaslané na jednotlivé servery. Nakonec se provede uvolnění paměti a návrat spojení s databází do `DBConnections Container`.

Nyní si rozebereme implementace metod reprezentující obsluhy jednotlivých zpráv v následujících podsekcích.

### 4.1.1 HandleAdapterMessage

`HandleAdapterMessage` je metoda obsluhující příchozí spojení od `Adapter Reciever server`u. Metoda nejprve využívá převzané třídy `ProtocolV1MessageParser` pro parsování

tohoto typu zprávy a ukládání dat z této zprávy do jejího objektu. Je to z toho důvodu, protože již před psaním této práce v rámci celého systému inteligentní domácnosti tento parser byl napsán pro `Adapter Server Reciever`, a proto nebylo důvod jej přepisovat. Dále metoda s daty uloženými do tohoto objektu pracuje.

Metoda nejprve vezme identifikátor adaptéru, ze kterého zpráva přišla a prostřednictvím metody `SelectIdsEnabledAlgorithmsByAdapterId` zadá dotaz nad databází, kde vyhledá všechny identifikátory povolených aplikačních modulů navázaných na uživatele v databázi. Následně pomocí těchto identifikátorů postupně vyhledává zařízení (senzory), které jsou na tyto uživatelské aplikační moduly navázány. Pokud dojde ke shodě identifikačního čísla ze kterého přišla právě data s číslem zařízení přiřazeném k uživatelskému algoritmu v databázi, tak se provede spuštění *aplikačního modulu* (viz. subsekce 4.1.6).

### 4.1.2 handleUIServerMessage

Je metoda obsluhující příchozí spojení od *UI serveru*. Nejprve se provede uložení potřebných informací ze zprávy, které vycházejí z komunikačního protokolu definovaného v návrhu.

V každé zprávě musí být minimálně obsažena verze protokolu (atribut `ver`, typ příchozí zprávy `state`, beeon token reprezentující řetězec identifikující komunikaci (kvůli principu komunikace dotaz-odpověď, kde se neudržuje stále spojení se serverem, atribut `bt`) a identifikační číslo uživatele (`userid`). Tyto veškeré informace jsou uloženy do proměnných nezávisle na typu příchozí zprávy.

Typů zpráv zasílaných na tento server je více, a pro každý z nich je implementována jiná obsluha, která přímo vyplývá z této zprávy. Ovšem na každou zprávu musí být implementována odpověď. Z toho důvodu při chybě ve zpracování zprávy (ať už je způsobena klientem či stranou *modulárního prostředí*) je pro všechny zprávy implementována možnost odeslat jako odpověď zprávu typu `false` s příslušným chybovým kódem.

Zde je tato chybová zpráva řešena takovým způsobem, že je vytvořena proměnná `error`, které je na počátku přiřazena pravdivostní hodnota `false`. Při vyskytnutí této potencionání chyby bude nastavena tato proměnná na `true` a tím se způsobí, že se nevytvoří standartní odpověď na každou zprávu, ale varianta chybová.

Následně se v samostatných blocích dle typu zprávy (dle `state`) provádí obsluha této zprávy a na konci obsluhy je vždy vytvořen řetězec k odeslání zprávy a uložen do proměnné `string To Send As Answer`. Až na konci celé metody `handleUIServerMessage` se tento text odpovědi předá metodě `sendMessageToSocket`, která již provede samotné odeslání zprávy.

Nyní si rozebereme jednotlivé implementace obsluhy těchto zpráv.

#### Zpráva addalg

Obsluha typu zprávy `addalg` přidá *aplikační modul* pro uživatele zasílající tuto zprávu do databáze. Implementace obsluhy zprávy `addalg` vychází z návrhu zprávy, který je uveden v kódu4.1.

```
<com
    ver="2.5"
    bt="42"
    state="addalg"
    algname="uživatelem definované jméno"
    aid="64206"
    atype="3" >
        <dev id="12345" type="1" pos="1"/>
        <par pos="1">42</par>
```

```
        <par pos="2">Text notifikace</par>
        <geo rid="478547" type="in" />
</com>
```

Kód 4.1: Zpráva typu `addalg`.

Nejprve se uloží příslušné informace ze zprávy, které jsou potřebné, do proměnných. Je to identifikační číslo adaptéru, na který má být navázán aplikační modul, číslo typu algoritmu a uživatelem definované jméno algoritmu.

Následně se vyvolá metoda `GetAlgorithmById` třídy `FrameworkConfig` (viz. podsekce 4.1.5), která se pokusí vyhledat specifikaci algoritmu zadanou v konfiguračním souboru *aplikačních modulů*. Pokud se specifikace nalezne, pak je možné tento *aplikační modul* přidat do databáze. Kontroluje se zde navíc zda je ve zprávě potřebný počet předaných parametrů a dat ze senzorů.

Pokud tedy proběhne kontrola v pořádku, pak se do struktury `tdevice` uloží ze zprávy informace o zařízeních (element `dev`) přiřazených k tomuto *aplikačnímu modulu*. Navíc z důvodu, že jich může být vyšší počet než jedna, tak se uloží do kolekce těchto struktur pro následné snadné zpracování při ukládání do databáze.

Podobné uložení proběhne s předanými uživatelskými daty k *aplikačnímu modulu* (elementy `par`, zde již definovanými pod termínem parametry. Tyto parametry se uloží do struktury `tparam` a následně do kolekce těchto struktur, protože jich může být více.

Jako poslední se ukládají elementy `geo` do struktury `tgeo` a následně do kolekce těchto struktur. Jsou určeny pro druhý typ aplikačních modulů vyvolávaný s příchodem zprávy `passborder`. Tento element nastavuje *modulárnímu prostředí* případ, kdy má *aplikační modul* spustit. Je to identifikační řetězec geolokační oblasti, ke které je navíc přidán atribut `type`, který určuje zda se má aplikační modul vyvolat při vstupu do této geolokační oblasti či při výstupu z ní.

Jakmile jsou korektně uloženy informace do kolekcí těchto tří struktur, pak je provedeno parsování parametrů (`par`) do řetězce k uložení do databáze ve formě textu. Je nutné dodat, že tento formát je již uložen takovým způsobem, jak se již předává *aplikačnímu modulu*, který jej zpracovává (jak se to řeší bude popsáno na konci této kapitoly).

Následně se uloží pomocí objektu třídy `DBFWHandler` a metody `InsertUserAlgorithm()` do databáze do tabulky `users_algorithms` tento *aplikační modul*. Pro úplnost je zde dodáno, že defaultní hodnota domény `state` je nastavena na 1, tedy při přidání *aplikačního modulu* uživatelem je zároveň povolen a spuštěn.

Práce s databází je také korektně ošetřena a pokud by nastala chyba, pak je nastavena do proměnné `error` hodnota `true` a následně je odeslána zpráva typu `false`.

Při správném zpracování databázového dotazu se vytvoří odpověď typu `AlgCreated`, která je definovaná v protokolu. (tato zpráva předává pouze identifikační číslo uživatelského aplikačního modulu vygenerovaného vložením nového řádku do tabulky `users_algorithms`.

### Zpráva getallalgs

Zpráva typu `getallalgs` je zpráva, kterou požaduje mobilní zařízení vypsání všech nastavených a vytvořených aplikačních modulů na daného uživatele. V kódu 4.2 je uveden její formát.

```
<com
    ver="x.x"
    bt="4442"
    state="getallalgs"
    aid="123"
/>
```

Kód 4.2: Zpráva typu `getallalgs`.

Obsluha této zprávy spočívá ve správných dotazech do databáze, protože se jedná o vyhledání v databázi a vypsání všech těchto vytvořených *aplikačních modulů* ve správném formátu.

K tomuto získání těchto vytvořených *aplikačních modulů* (v rámci příchodů dříve popsané zprávy `addalg`) zde byla implementována metoda getAllAlgsByAdapterIdAndUserId. Tato metoda navrátí kolekce (`vector<talg *>`) struktur obsahující informace o všech těchto aplikačních modulech a následně je předána do metody `createMessageAlgs`, která již vygeneruje potřebný formát zprávy (`algs`) pomocí této kolekce a je následně předána k odeslání. Níže vidíme implementaci struktury `talg`

```
typedef struct alg
{
    std::string usersAlgorithmsId;
    std::string atype;
    std::string enable;
    std::string name;
    std::vector<tchldMsg*> childs;
} talg;
```

Kód 4.3: Struktura `talg`.

Pro úplnost a znovupoužitelnost je metoda `getAllAlgsByAdapterIdAndUserId` rozdělena do ještě další metody `getAlgByUserAlgorithmId`, která navrátí naplněnou strukturu. Je využita u následující zprávy.

### Zpráva getalgs

Zpráva `getalgs` obsahuje požadavek na odeslání *aplikačních modulů* dle jejich primárního klíče v databázi. Formát přijaté zprávy z mobilního zařízení je popsán kódem 4.4.

```
<com
    ver="x.x"
    bt="44"
    aid="64206"
    state="getalgs">
    <alg id="123"/>
    <alg id="143"/>
</com>
```

Kód 4.4: Zpráva typu `getalgs`.

Implementace obsluhy zprávy je podobná jako u popisu minulé obsluhy zprávy s jedniným rozdílem, že se nevyhledávají všechny *aplikační moduly* přiřazené k jednomu uživateli, ale vyhledává se pouze dle jejího primárního klíče. Proto můžeme využít již vytvořené a popsané metody `getAlgByUserAlgorithmId`, která nám vrátí strukturu `talg` 4.3, kterou pouze vložíme do kolekce s jedním prvkem této struktury a předáme metodě `createMessageAlgs` stejně jako v předchozí popsané zprávě.

### Zpráva setalg

Zpráva typu `setalg` požaduje z mobilního zařízení uživateli pozměnit chování *aplikačního modulu* (resp. jeho nastavení). Formát zprávy ze zařízení je velice podobný zprávě `addalg`, který můžeme vidět v kódu 4.5.

```
<com
      ver="x.x"
      bt="4415442"
      state="setalg"
      algid="123"
      algname="algor"
      enable="1"
      atype="3">
          <dev id="12345" type="1"/>
          <par pos="1">25</par>
          <par pos="2">Pozměněný text notifikace</par>
          <geo rid="478547" type="in" />
</com>
```

Kód 4.5: Zpráva typu `setalg`.

Implementace obsluhy tohoto typu zprávy se liší pouze v typu databázovém dotazu. Zde se nejprve zase předají potřebné informace do proměnných a struktur a následně se již dle `algid` pouze provede `UPDATE` řádku tabulky s daným id v tabulce `users_algorithms`. Aktualizace přiřazených zařízení k *aplikačnímu modulu* se provádí tím způsobem, že se nejdříve odstraní z databáze v tabulce `algo_devices` všechny zařízení přiřazené k tomuto *aplikačnímu modulu* a následně se dle nové definice uživatelského *aplikačního modulu* vytvoří znovu potřebné řádky v této tabulce.

Pokud je zpráva ve správném formátu a databázové dotazy proběhnou v pořádku, pak je pro odpověď vytvořena zpráva typu `true` (metoda `createMessageTrue()`.

### Zpráva delalg

Zpráva typu `delalg` obsahuje požadavek o smazání *aplikačního modulu* přiřazeného k uživateli. Formát zaslané zprávy je zde popsán v kódu 4.6.

```
<com
      ver="x.x"
      bt="4415442"
      state="delalg"
      algid="123"
/>
```

Kód 4.6: Zpráva typu `delalg`.

Implementace obsluhy této zprávy spočívá v získání identifikátoru uživatelského *aplikačního modulu* ze zprávy (`algid`), ve smazání zařízení přiřazených k tomuto *aplikačnímu modulu* z databáze pomocí metody `DeleteAlgoDevices()` a nakonec smazání samotného uživatelského aplikačního modulu z databáze z tabulky `users_algorithms` (metoda `DeleteUsersAlgorithms()`).

Pokud vše proběhne v pořádku, pak je pro odpověď na mobilní zařízení vytvořena zpráva typu `True` pomocí metody `createMessageTrue()`.

#### Zpráva passborder

Zpráva typu `passborder` je zpráva, která se odešle z mobilního zařízení při překročení hranice geolokační oblasti. Je to spouštěcí mechanismus druhého typu *aplikačních modulů* (typ spouštěný zasláním zprávy z mobilního telefonu). Formát zprávy je popsán v kódu 4.7.

```
<com
      ver="x.x"
      bt="4415442"
      state="passborder"
      rid="home"
      type="in"
/>
```

Kód 4.7: Zpráva typu `passborder`.

Obsluha této zprávy nejprve zjistí, zda jsou k danému uživateli na daný typ *aplikačního modulu* vytvořeny nějaké uživatelské *aplikační moduly*. Provede se tedy databázový dotaz pomocí metody `SelectIdsAlgorithmsByAlgIdAndUserId()`, který navrátí seznam identifikačních čísel reprezentující primární klíče tabulky `users_algorithms`.

Pokud výsledek databázového dotazu byl neprázdný, pak již pro každý vyhledaný záznam se spustí jeden *aplikační modul*, kdy každému z nich jsou předána stejná data ze zprávy (identifikační číslo geolokační oblasti `rid` a směr zda se do oblasti vstoupilo, resp. vystoupilo `type`). Jakým způsobem je řešeno spouštění *aplikačních modulů* bude vysvětleno dále v textu.

Pokud jsou všechny databázové dotazy provedeny v pořádku, pak je pro odpověď vytvořena zpráva typu `true` pomoc metody `createMessageTrue()`.

### 4.1.3 HandleAlgorithmMessage

Metoda `HandleAlgorithmMessage` implementuje obsluhu zpráv zasílaných od *aplikačních modulů*. Tyto zprávy předávají informace pro identifikaci jaký *aplikační modul* (navázaný na jakého uživatele, adaptér atp.) zprávu odeslal se seznamem požadavků na tento server pro vykonání nad celým systémem.

*Aplikačnímu modulu* byly implementovány dvě podpory pro práci nad systémem. A to možnost zaslání notifikační zprávy na mobilní zařízení s náležitou identifikací a parametrizací této zprávy. A možnost změnit stav aktoru na straně adaptéru.

Z této podpory byl následně navržen formát XML zprávy, který je od aplikačních modulů očekáván, viz kód 4.8.

```
<alg_m protocol_version="1.0" userID="7" algID="1"
adapterID="5544">
    <notifs count="1">
        <notif type="1" text="Text notifikace" />
    </notifs>
    <tactors count="1">
        <tactor id="429" type="2" />
    </tactors>
</alg_m>
```

Kód 4.8: Zpráva `alg_m` zaslaná z *aplikačního modulu*.

Ze zprávy vidíme, že jsou zde dva synovské elementy `notifs` a `tactors`. Tyto elementy reprezentují jednotlivé dvě podpory, které byly zmíněny výše. Tyto dva synovské elementy mají jako atribut `count`. Dle těchto atributů metoda zjistí, zda má provádět nějakou z požadovaných akcí a kolik těchto akcí daného typu bude.

Pokud je ve zprávě požadavek na odeslání notifikační zprávy na telefon, pak se provede databázový dotaz pomocí metody `GetNotifStringByUserId()`, který vrátí všechny identifikační řetězce mobilních zařízení, který vlastní daný uživatel, který si zažádal odeslat zprávu (resp. který si vytvořil a povolil aplikační modul ve svém telefonu). Pokud je v databázi nalezen alespoň jeden takový záznam s takovým řetězcem reprezentujícím mobilní zařízení, pak je následně naplněn konstruktor třídy `WatchdogNotif` a vytvořen jeho objekt. Následně se zavolá metoda `sendGcm()` nad tímto objektem, který se již postará o odeslání zprávy. Také je zde řešena návratová hodnota této metody `sendGcm()`, která vrací kolekci řetězců identifikující mobilní zařízení, na které se zpráva neodeslala. To je ošetřeno proto, aby se na toto mobilní zařízení již nepokoušelo znovu zprávy odesílat, tak je o něm smazán záznam v databázi (tabulka `mobile_devices`) pomocí metody `DeleteMobileDeviceByGcmId()`. Následně je o odeslání notifikační zprávy uložen záznam v databázi pomocí metody `InsertNotification()`.

V případě presence požadavku na změnu aktoru (či více aktorů) je načten každý požadavek v cyklu zvlášť a pro každý tento požadavek je provedeno vytvoření správného formátu zprávy (viz kód 4.9) pomocí metody `createMessageRequestSwitch()`.

```
<request type="switch">
    <sensor id="1.1.1.1" type="0x00" onAdapter=12345">
        <value>ON</value>
    </sensor>
</request>
```

Kód 4.9: Zpráva switch na *Ada Server Sender*.

A následné odeslání zprávy na příslušný aktor pomocí metody `sendMessageToAdaServer()`. Metoda `sendMessageToAdaServer()` pracuje tím způsobem, že zasílá tuto XML zprávu na `Adapter Server Sender`, který zprávu přeposílá dále na fyzické zařízení. Vzhledem k tomu, že `Ada Server Sender` posílá nazpět zprávu, zda bylo přijata zpráva ve správném formátu a zda jej korektně odeslal dále, tak zaznamenává do logovacího souboru případné chyby, které se dají lépe opravit. Lépe se zjistí, že server není v provozu (timeout) a podobně.

### 4.1.4 Práce s databází

Pro práci s databází zde byla využita externí knihovna SOCI (Sobczak et al.), která podporuje snadný přístup k databázi a je implementována pro jazyk C++.

Aby bylo možné celému hlavnímu programu *modulárního prostředí* poskytnout spojení s databází (tzv. sezení), tak byla implementována třída `DBConnectionsContainer`. Tato třída při její instanciaci vytvoří více těchto sezení a je schopna následně předávat celé aplikaci v případě potřeby tyto spojení a v okamžiku, kdy jej nějaká z části aplikace (např. obsluha zprávy)) již nebude potřebovat, tak jej navrátí zpět do tohoto kontejneru. Řeší se tím problém stálého připojování a odpojování k/od databáze, což by velmi zpomalovalo proces vykonání databázového dotazu. Počet těchto sezení vytvořených v kontejneru se nastavuje v rámci konfiguračního souboru popsaného výše.

Konkrétně tedy instance třídy `DBConnectionsContainer` vydá pomocí metody `GetConnection()` spojení (session), což je již objekt zmíněné externí knihovny SOCI. Aby se s tímto spojením dalo pohodlně pracovat, je zde implementována třída `DBFWHandler`, do jejíhož

objektu je toto sezení předáno do konstruktoru a uloženo v rámci něj. Tato třída `DBFWHandler` implementuje již metody, které nad objektem z třídy `session` provádí databázové dotazy. Níže uvádím příklad jedné z implementací jejích metod na provedení databázového dotazu.

```
1  std::string DBFWHandler::SelectUserIdByUsersAlgId(
2  std::string UsersAlgId)
3  {
4      std::string retVal = "";
5      this->_log->WriteMessage(TRACE, "Entering " + this->_Name +
6       "::SelectAlgIdByUsersAlgId");
7      std::string sqlQuery = "SELECT fk_user_id FROM " +
8      "users_algorithms WHERE users_algorithms_id = " + UsersAlgId
9       + ";";
10     this->_log->WriteMessage(TRACE, sqlQuery);
11     try
12     {
13         *ses << sqlQuery, into(retVal);
14     }
15     catch (std::exception const &e)
16     {
17         std::string ErrorMessage = "Database Error : ";
18         ErrorMessage.append(e.what());
19         this->_log->WriteMessage(ERR, ErrorMessage);
20         retVal = "0";
21     }
22     this->_log->WriteMessage(TRACE, "Exiting " + this->_Name +
23     "::SelectAlgIdByUsersAlgId");
24     return (retVal);
25 }
```

Kód 4.10: Metoda třídy DBFWHandler.

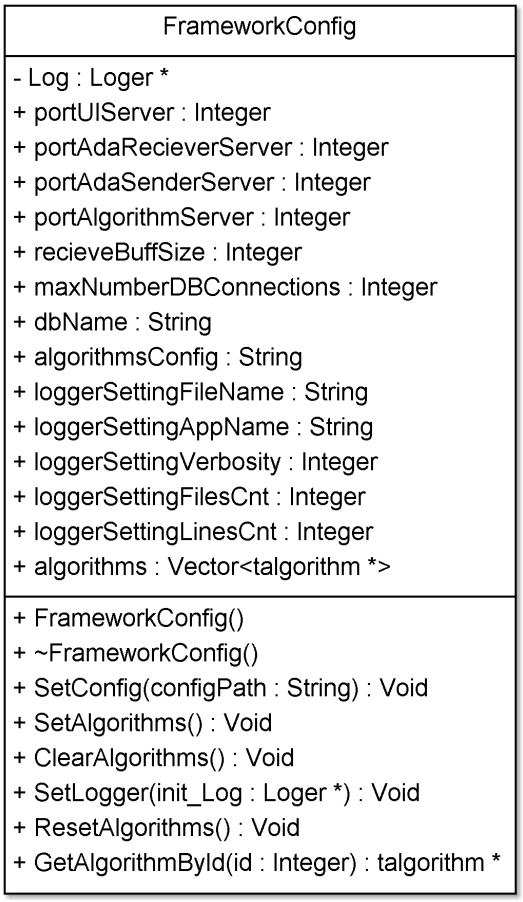
Výstupy těchto metod závisí na typu databázového dotazu. Při SQL dotazu `INSERT` je vracen buď nově vytvořený primární klíč (index) nebo pouze pravdivostní hodnota o správném provedení. Při dotazech `SELECT` vrací metody buď seznamy řetězců (pokud se jedná o výstup více hodnot), jeden řetězec, celé číslo nebo reálné číslo. Při dotazech `DELETE` vrací metody pouze pravdivostní hodnotu o jejím úspěšném (resp. neúspěšném) smazání.

Pokud dojde k chybě v průběhu provádění dotazu, pak je chyba zaznamenána také do logovacího souboru.

### 4.1.5 Konfigurátor

Pro konfiguraci hlavní aplikace *modulárního prostředí* je implementována třída `FrameworkConfig`. Definice této třídy popisuje UML diagram její třídy na obrázku 4.3.

Nejprve zavoláme konstruktor této třídy k vytvoření objektu a teprve následně je volána metoda `SetConfig()` v rámci tohoto objektu, které je předána cesta k XML souboru s konfigurací. Metoda `SetConfig()` zpracuje tento XML soubor (pomocí dříve zmíněné knihovny pugixml) a uloží do atributů objektu nastavení zadané v tomto konfiguračním souboru. Jak vidíme z definice třídy, tak definice *aplikačních modulů* jsou uloženy jako kolekce struktur `talgorithm`. Tyto




Obrázek 4.3: Třída FrameworkConfig.

definice *aplikačních modulů* je umožněno znovu nastavit v průběhu běhu aplikace pomocí metody `ResetAlgorithms()`. To zde bylo implementováno z toho důvodu, že dle zadání bylo nutné implementovat takové prostředí, do kterého je možné vkládat a také odebírat aplikační moduly. Jedna z mých interpretací této věty byla, že je možné vkládat přímo i jejich zdrojové kódy, neboli celé *aplikační moduly* do *modulárního prostředí*. Pro úplnost vkládání aplikačních modulů je zde řešeno i jako přiřazování a povolování jejich spouštění přiřazené k jednomu uživateli při příchodu senzorických dat či při příchodu zprávy z mobilních zařízení (momentálně `passborder`). Proto aby bylo možné vložit do celého *modulárního prostředí aplikační modul*, pak je nutné zadat k jeho definici cestu v konfiguraci definic (seznamu definic) *aplikačních modulů*. Jakmile to uživatel udělá, pak je implementován reset seznamu těchto definic pomocí zaslání signálu `SIGINT` do aplikace *modulárního prostředí*, které tento signál korektně odchytí a provede znovunačtení tohoto seznamu *aplikačních modulů*.

### 4.1.6 Spouštění aplikačních modulů

Než vysvětlím implementaci jak probíhá spouštění aplikačních modulů, tak je třeba si znovu rozebrat zadání této práce. Cílem návrhu této práce je navrhnout *modulární prostředí*, které by mělo umožňovat zpracování senzorických dat pomocí vložení samostatných *aplikačních modulů*, které se mohou zavádět, spouštět, pozastavovat atd.

Můj původní návrh byl implementovat toto prostředí takovým způsobem, že bude tyto aplikační

moduly spouštět jako vlákna a bude jim předávat data ve formě objektu. Ovšem z důvodu znění zadání, kde je psáno, že musí práce umožňovat „vložení samostatných aplikačních modulů“, bylo nutné zvolit jiný scénář. A to vytvořit při spouštění nový proces pomocí přeloženého programu, kterému říkáme *aplikační modul*. Takto je možné následně při běhu *modulárního prostředí* zadat pouze do konfiguračního souboru (novou) cestu k tomuto spustitelnému programu (s počtem jeho potřebných parametrů, dat ze zařízení atd.) a načíst pomocí signálu `SIGINT` znovu tyto *aplikační moduly*. Těmito dvěma kroky můžeme vložit nový aplikační program do *modulárního prostředí* až za jeho běhu, což je cílem této práce.

Spuštění aplikačního modulu znamená provedení funkcí `fork()` a následně v programu potomka zavolání funkce `execvp()` se zadáním názvu binárního programu a zadáním argumentů jemu předaných při spuštění. Klíčový je zde formát předaných argumentů do *aplikačního modulu*.

Argumenty předané do programu jsou následující `-u` předává uživatelské identifikační číslo (z databáze), `-a` předává identifikační číslo typu algoritmu, `-d` předává identifikační číslo adaptéru pro který je aplikační modul spouštěn, `-o` předává identifikační číslo reprezentující jedinečný identifikátor uživatelského algorimtu (tabulka `users_algorithms`), `-v` předává přijatá data ze senzorů (nebo o `passborder` z telefonu) na základě kterých se spouští aplikační modul. Data předaná v tomto argumentu `-v` je nutné dodržet v následujícím formátu `RidOrDevicedevice#ID44...` V argumentu `-p` se předávají parametry zadané uživatelem při ukládání do databáze (již o nich byla zmínka u obsluhy zprávy z UI serveru při typu zprávy `addalg`). Tyto parametry musí mít formát „`16964877---10#gt#0#notif#Text notifikace`“, kde # je oddělovač jednotlivých parametrů a text parametru musí být dlouhý alespoň jeden znak. A posledním argumentem je `-e`, který předává název lokální databáze, se kterou má *aplikační modul* pracovat.

### 4.1.7 Zpracování signálů

Aplikace zpracovává korektně dva typy signálů. A to signály `SIGINT` a `SIGTERM`.

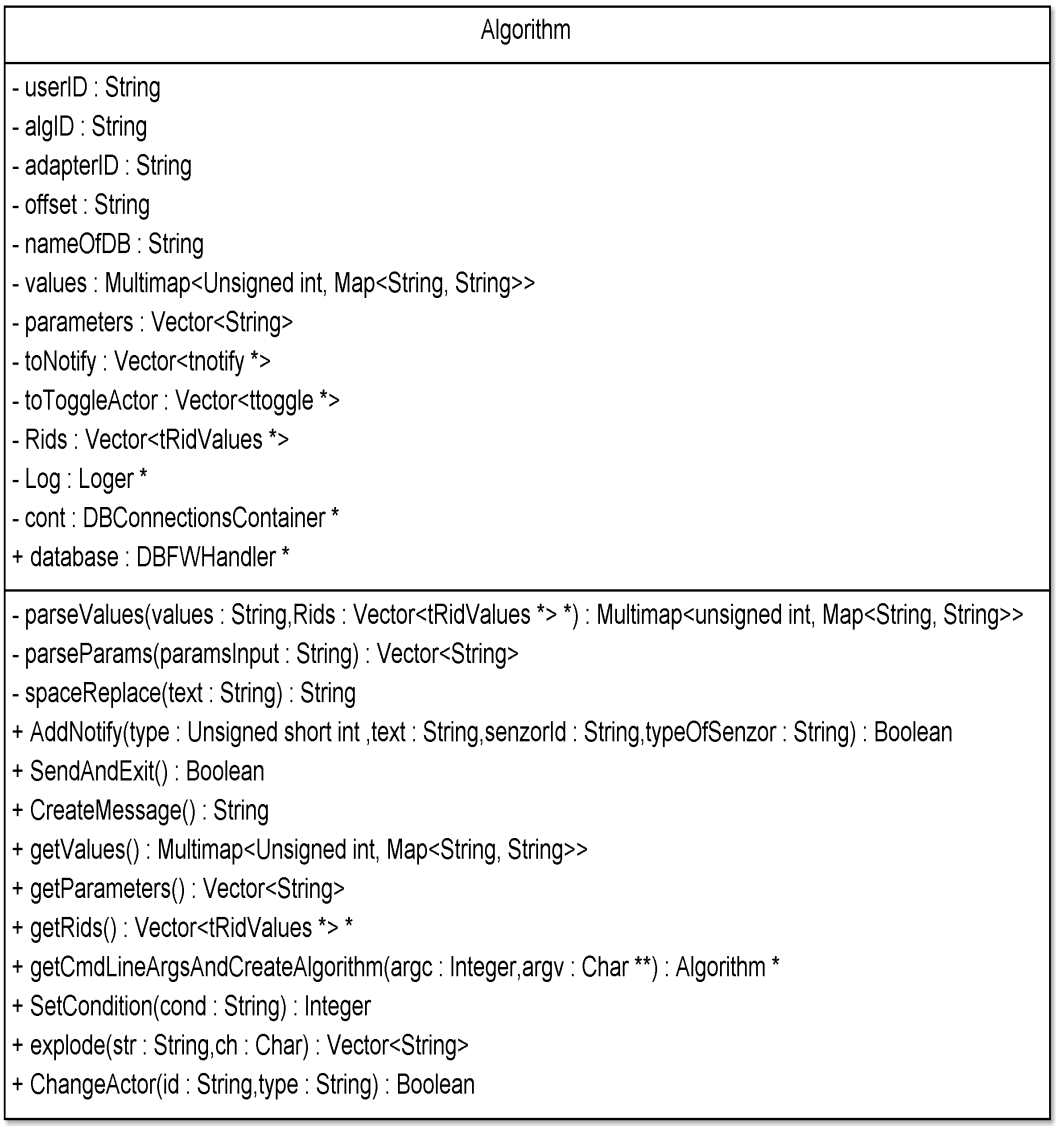
Při přijetí signálu `SIGINT` program vyvolá nad objektem třídy `FrameworkConfig` metodu `ResetAlgorithms()`, která znovu načte definice *aplikačních modulů* (cesty k jejím spustitelným souborům atd.) a uloží do tohoto objektu. Zpracování signálu `SIGTERM` spočívá v korektním ukončení celého programu. Ukončení všech tří serverů, uvolnění paměti a uvolnění databázového spojení.

## 4.2 Implementace třídy pro tvorbu Aplikačního modulu

*Aplikační modul* je navržen jako samostatný spustitelný program v příkazové řádce v linuxovém prostředí. Pro jeho tvorbu je poskytnuta třída `Algorithm`, viz obrázek 4.4.

Tato třída již poskytuje rozhraní pro komunikaci s *modulárním prostředím* bez nutné znalosti její implementace. Pro její instanciaci je zde vytvořena statická metoda `getCmdLineArgs And Create Algori thm()`, která převezme argumenty příkazového řádku a pokud vše zpracuje správně (není zde chyba ve formátu argumentů a podobně), pak vrací objekt třídy `Algorithm`, se kterým autor *aplikačního modulu* pracuje. Očekávávané argumenty při spuštění jsou definovány v podsekci 4.1.6.

Jakmile je vytvořena instance této třídy, je z ní umožněno autorovi *aplikačního modulu* získat předaná data pomocí metod `getParameters()` (parametry zadané uživatelem), `getRids()` (geolokační oblasti se směrem o překročení) a `getValues()` (konkrétní hodnota dat přijatá od senzoru). Je nutné dodat, že autor *aplikačního modulu* sám ví (z definice aplikačního modulu na stránkách inteligentní domácnosti a v konfiguračním souboru) kolik parametrů očekává, a také jen on zná přesnou sémantiku parametru na dané přijaté pozici. Z toho důvodu je opravdu nezbytně nutné




Obrázek 4.4: Třída Algorithm.

dodržovat formát definice pořadí zaslaných parametrů, protože by mohlo dojít k nepředvídanému chování celého *aplikačního modulu*.

Nad těmito daty může *aplikační modul* provádět výpočty a v rámci nich má navíc k dispozici již předem připravené spojení s databází (`session` externí knihovny `SOCI`). Toto spojení je vytvářeno již při vytváření objektu třídy `Algorithm`, do kterého je vložen ukazatel na objekt třídy `DBFWHandler` (atribut `database`). Pomocí tohoto objektu má autor *aplikačního modulu* k dispozici jakýkoli dotaz implementovaný v rámci třídy `DBFWHandler`. V průběhu výpočtů (kdekoli v programu) může autor využít akce nad celým systémem inteligentní domácnosti, které jsou poskytnuty dvě. Notifikace na mobilní zařízení a změna aktoru pomocí metod `AddNotify()` a `ChangeActor()` implementovaných ve třídě `Algorithm`.

Na konec každého *aplikačního modulu* je nutné vždy zavolat metodu `SendAndExit()`, která se již postará o odeslání veškerých požadavků na server hlavního programu *modulárního prostředí*. Pro úplnost je zde uveden vzorový příklad kódu jednoduchého *aplikačního modulu*.

```
1  #include <cstdlib>
2  #include "../algorithm.h"
3
4  using namespace std;
5
6  int main(int argc, char *argv[])
7  {
8      //Deklarace a inicializace objektu Algorithm
9      //poskytující rozhraní pro tvorbu algoritmu.
10     Algorithm *alg;
11     if ((alg = Algorithm::getCmdLineArgsAndCreateAlgorithm(
12     argc, argv)) == nullptr)
13     {
14         return EXIT_FAILURE;
15     }
16     /*Tělo programu*/
17
18     /*Konec těla programu*/
19     //Odeslání dat do Frameworku a ukončení algoritmu.
20     if (!alg->SendAndExit()){
21         delete(alg);
22         return EXIT_FAILURE;
23     }
24     delete(alg);
25     return EXIT_SUCCESS;
26 }
```

Kód 4.11: Vzorový příklad kódu *aplikačního modulu*.

# Kapitola 5

# Nasazení a výsledky

V této kapitole je popsán způsob testování aplikace a její nasazení do již reálného systému, pro který byla vyvíjena, včetně zhodnocení výsledků její funkčnosti.

## 5.1 Testování

Před nasazením *modulárního prostředí* do systému inteligentní domácnosti se provádělo testování, které probíhalo v několika rovinách.

Vývoj byl prováděn pomocí metodiky programování řízeného testy (TDD), podle (Beck, 2004). Dle návrhu byly vytvořeny jednotkové testy, pomocí nichž byly testovány jednotlivé třídy programu. K tomuto přístupu byl využit nástroj Boost.Test [1], což je framework pro testování aplikací psaných v jazyce C++. Byl zde vybrán pro svou snadnou instalaci (jeho knihovny lze nainstalovat na linuxové prostředí ve formě instalačního balíku) a použitelnost.

Systémové testy byly prováděny pomocí konceptu černé skříňky (tzv. black box), který je rozebraný například v publikaci (Patton, 2002). Jedná se o dynamické testování aplikace bez znalosti její implementace, na základě sad vstupních dat a k nim adekvátních dat výstupních. Vstupy byly předávané do aplikace *modulárního prostředí* pomocí simulovaných zpráv zasílaných na aplikaci *modulárního prostředí*. Na základě těchto zpráv se do logovacího souboru vypisovaly záznamy, podle kterých se hodnotilo správné vykonání reakce na tyto zprávy. Při určitých zprávách (testech) se také vytvářely záznamy v databázi, kde se hodnotilo, zda opravdu vznikly a zda obsahovaly očekávané hodnoty. Pro otestování správné funkčnosti třídy `Algorithm` pro tvorbu *aplikačních modulů* bylo implementováno několik jednoduchých variant *aplikačních modulů*. Ke každé variantě byla napsána sada vstupních a výstupních testů, kde vstupy byly zadávány jako argumenty programu příkazové řádky a výstupy byly srovnávány s očekávanými výpisy v logovacím souboru *modulárního prostředí*.

## 5.2 Nasazení

Nasazení, neboli integrace *modulárního prostředí* do reálného provozu, bylo složeno z následujících kroků:

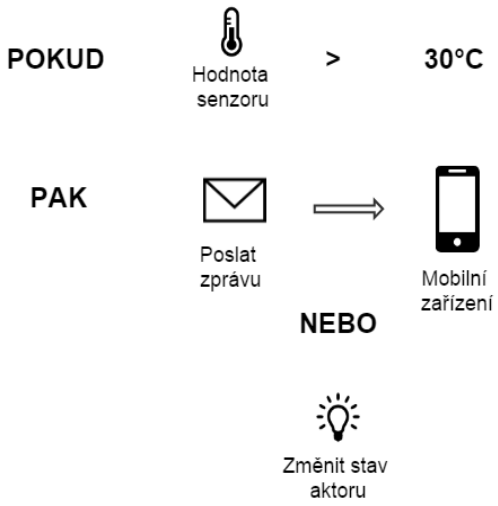
- umístění zdrojových kódů aplikace na server (existujícího systému),
- přeložení aplikace *modulárního prostředí*,

[1] http://www.boost.org/doc/libs/1_43_0/libs/test/doc/html/utf.html

- přeložení *aplikačních modulů*,
- nastavení správné konfigurace v konfiguračním souboru (např. jméno databáze, cesta k definicím aplikačních modulů),
- nastavení konfigurace v konfiguračním souboru *aplikačních modulů* (např. cesta k přeloženým spustitelným aplikacím),
- spuštění *modulárního prostředí*,
- kontrola správného spuštění dle výpisů v logovacím souboru,
- testování správnosti komunikace se zbytkem systému,
- běh *modulárního prostředí* v rámci inteligentní domácnosti.

Pomocí této integrace se podařilo objevit několik drobných chyb, které byly následně odstraněny. Pro *modulární prostředí* jsem v rámci nasazení (nad rámec této práce) implementoval dva *aplikační moduly* (jejich popis níže), aby se *modulární prostředí* začalo používat k praktickému použití.

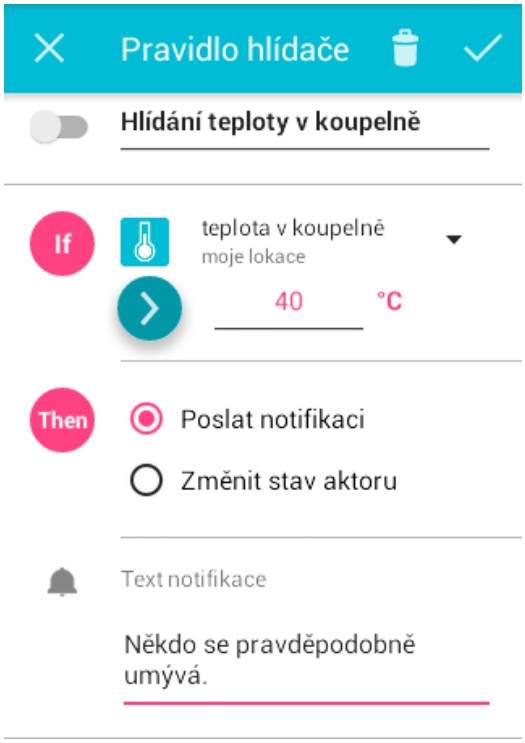
Aplikace byla v tento okamžik zhodnocena jako funkční, a proto bylo vhodné pro ni vytvořit nějaký *aplikační modul* k reálnému použití. Jako první byl navržený *aplikační modul* s názvem *Watch and notify*. `Watch and notify` je *aplikační modul* prvního typu (spouštěn na základě příchodu dat ze senzoru) a jeho funkcí je hlídání hodnoty konkrétního aktoru, která když splní danou podmínku, tak se vykoná určitá akce. Uživatel si na mobilním telefonu zvolí senzor, který chce hlídat, operátor (`>`, `<`, `=`...) a hodnotu jako pravou stranu podmínky. Pokud je tato podmínka splněna, může si zvolit jaký text chce odeslat na mobilní zařízení nebo jaký aktor má přepnout svůj stav. Na obrázku 5.1 je popsán zcénář použití tohoto *aplikačního modulu* a na následujících obrázcích 5.2, 5.3 je ukazaná přímá práce s tímto aplikačním modulem v praxi na mobilním zařízení.



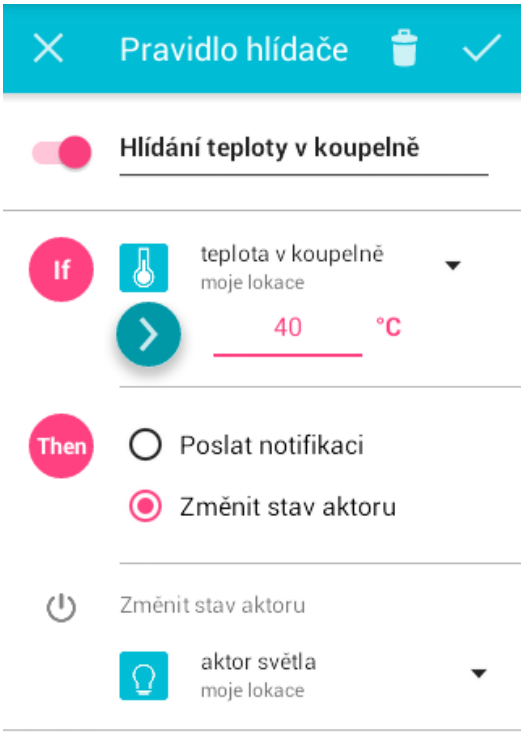

Obrázek 5.1: Scénář použití *aplikačního modulu Watch and notify*.

Praktický příklad využití *Watch and notify* je hlídání teploty v místnosti, kdy při překročení 30 stupňů Celsia sepneme spínač klimatizace. Definice jeho vstupních parametrů (dle popisu v kapitole 3) je uvedena v tabulce 5.1.

Do *modulárního prostředí* po nasazení také již korektně přicházely zprávy ke spuštění druhého typu *aplikačního modulu*. Proto byl pro obsluhu těchto zpráv vytvořen druhý aplikační modul s




Obrázek 5.2: Ukázka nastavení *aplikačního modulu Watch and notify* uživatelem v mobilním zařízení.




Obrázek 5.3: Ukázka nastavení *aplikačního modulu Watch and notify* s podmíněnou změnou aktoru.

názvem *Geofencing*. Scénář užití tohoto *aplikačního modulu* je vyobrazen na obrázku 5.4. Jeho funkcí je hlídání, zda uživatel nepřekročil tzv. geolokační oblast (oblast na mapě) v daném směru, v jakém si hlídání definoval. Uživatel nejdříve na mobilním zařízení definuje na mapě oblast, pro kterou chce hlídat (Obrázky 5.5, 5.6 a 5.7). Následně vytvoří aplikační modul *Geofencing*, kterému předá identifikační číslo této oblasti a zadá směr, který chce hlídat (Obrázek 5.8). Jako akci při zachycení správného přechodu z/do této geolokační oblasti může uživatel nastavit zaslání textu notifikace na mobilní telefon, či jaký aktor má přepnout jeho stav (podobně jako v aplikačním modulu *Watch and notify*). Definice vstupních parametrů pro *aplikační modul Geofencing* je uveden v tabulce 5.2.

Implementací těchto dvou *aplikačních modulů* se zároveň ověřil hlavní cíl *modulárního prostředí*, které je schopné poskytnout prostředí pro vkládání takovýchto *aplikačních modulů*, které počítají nad daty a vykonávají akce. Systém je během psaní této práce nasazený a je spuštěný na stroji s těmito technickými specifikacemi, kde pracuje správně a má své reálné využití:

- základní deska **Supermicro X7DB8 Motherboard**,
- procesor **Intel® Xeon® CPU Quad Core E5410 2.33GHz**,
- velikost operační paměti RAM **10 GB**,
- operační systém **CentOS 6.5**.

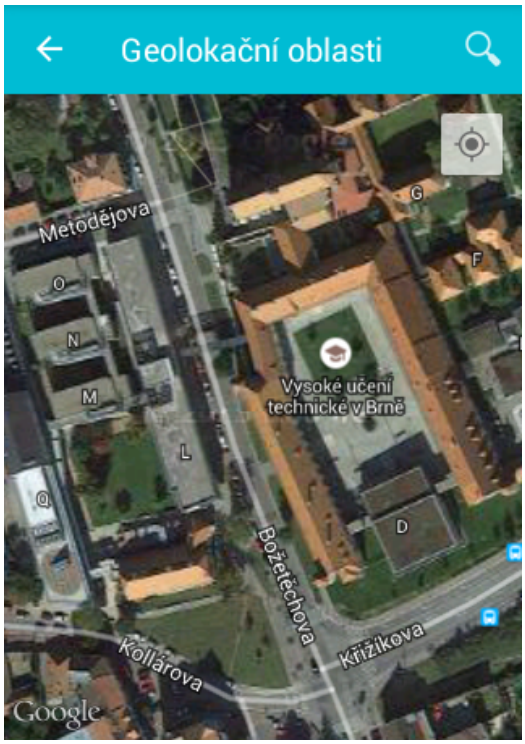
| Watch_and_notify | 1 |
|---|---|
| | |
| SENZOR_VALUE | ANY_SENZOR_VALUE |
| OPERATOR | ENTITY |
| VALUE | REAL |
| NOTIF_OR_ACTOR | NOTIF_OR_ACTOR |
| TEXT NOTIFIKACE NEBO ID AKTORU | STRING |

Tabulka 5.1: Definice *aplikačního modulu Watch and notify*.

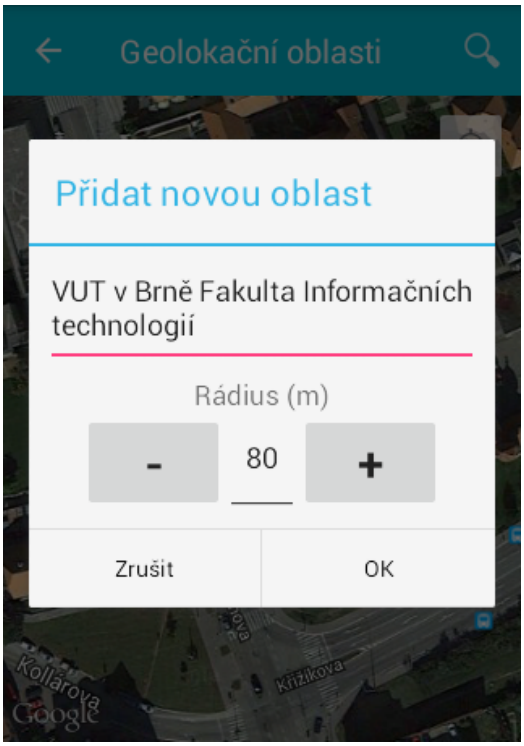

Obrázek 5.4: Scénář použití *aplikačního modulu Geofencing*.

| Geofencing | 3 |
|---|---|
| | |
| RID | STRING |
| TYPE | DIRECTION |
| NOTIF_OR_ACTOR | NOTIF_OR_ACTOR |
| TEXT NOTIFIKACE NEBO ID AKTORU | STRING |

Tabulka 5.2: Definice *aplikačního modulu Geofencing*.

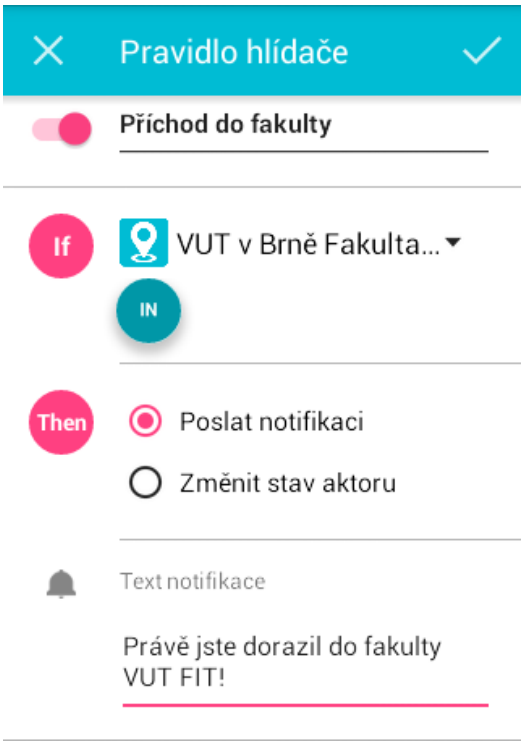



Obrázek 5.5: Obrazovka pro definování místa pro přidání *Geolokační oblasti*.




Obrázek 5.6: Obrazovka pro zadání údajů o nové *Geolokační oblasti*.


Obrázek 5.7: Ukázka obrazovky s již přidanou *Geolokační oblastí*.




Obrázek 5.8: Ukázka přidání a nastavení aplikačního modulu *Geofencing*.

# Kapitola 6

# Závěr

V bakalářské práci jsem navrhl a implementoval systém pro zpracování senzorických dat s podporou vkládání samostatných aplikačních modulů v rámci výzkumné skupiny ANT [1] a její podskupiny zaměřené na IOT [2]. Celý systém tak nabývá podoby modulárního prostředí (frameworku), které poskytuje rozhraní pro tvorbu dalších aplikačních modulů, bez nutné znalosti práce tohoto prostředí.

Fázi návrhu předcházelo studium dosavadního systému inteligentní domácnosti, kde jsem rozebral architekturu systému, seznámil jsem se s klíčovými fakty o všech částech systému a následně se zaměřil na serverovou část, pro kterou i návrh mého řešení byl určen. Na této serverové části jsem se seznámil jakým způsobem zde přicházejí data, jak je již navržena databáze a jakým způsobem bude možno komunikovat s mobilními zařízeními. Podrobnosti jsou uvedeny v kapitole č. 2.

Na základě poznatků z teoretické části jsem provedl návrh nových tabulek do schématu databáze pro ukládání dat modulárního prostředí, navrhl jsem rozhraní pro komunikaci se zbytkem systému a vytvořil návrh pro konfiguraci prostředí pro co nejsnazší nasazení na více serverových stanicích. Následně byly rozebrány požadavky na co nejobecnější budoucí použití aplikačních modulů a na základě nich byla rozebrána data nutná pro poskytnutí těmto modulům ze strany modulárního prostředí.

Dle návrhu byla provedena následná realizace řešení projektu s rozšířením o možnost definovat druhý typ aplikačního modulu spuštěný při příchodu požadavku o toto spuštění z mobilního zařízení. Na konci kapitoly implementace jsem detailně popsal jakým způsobem je řešen aplikační modul s návodem, jak si budoucí autor bude moci napsat svůj vlastní aplikační modul, který bude moci být spouštěný prostředím.

Na konci práce jsem popsal způsob testování aplikace a vzhledem k tomu, že již byla aplikace integrována do systému inteligentní domácnosti, tak jsem zde rozebral postup nasazení na fyzický server. Pro toto nasazení jsem navíc navrhl dva aplikační moduly, které jsem popsal a následně realizoval jako důkaz možného využití tohoto prostředí, které již má své praktické využití.

Možný další rozvoj aplikace, vytvořené v rámci této bakalářské práce, vidím v jejím rozšíření o možnost definovat další typy aplikačních modulů, například pro možné definování aplikačního modulu spouštěného periodicky po určitém čase, nikoliv získávající data přímo z aktuálních příchozích dat ze senzorů, ale z historických dat uložených v databázi. Tímto způsobem by se mohly provádět pravidelné výpočty, například pro předpověď počasí nebo by to umožnilo uživateli definovat přesnou dobu zapnutí centrálního vytápění objektu.

[1] http://merlin.fit.vutbr.cz/ant/

[2] Internet Of Things

# Literatura


BECK, K. *Programování řízené testy*. 1. vyd. Praha : Grada Publishing, a.s., 2004. ISBN 80-7226-636-5.

KAPOULKINE, A. *pugixml* [online]. 2006. [cit. 15. 5. 2015]. Dostupné z: http://cdn.rawgit.com/zeux/pugixml/v1.5/docs/quickstart.html.

KOSEK, J. *XML pro každého*. 1. vyd. Praha : Grada Publishing, a.s., 2000. ISBN 80-7169-860-1.

MATOUŠEK, P. *Síťové aplikace a jejich architektura*. 1. vyd. Brno : VUTIUM, 2014. ISBN 80-2143-766-9.

PATTON, R. *Testování softwaru*. 1. vyd. Praha : Computer Press, 2002. ISBN 80-7226-636-5.

SOBCZAK, M. et al. *SOCI* [online]. [cit. 15. 5. 2015]. Dostupné z: http://soci.sourceforge.net/.

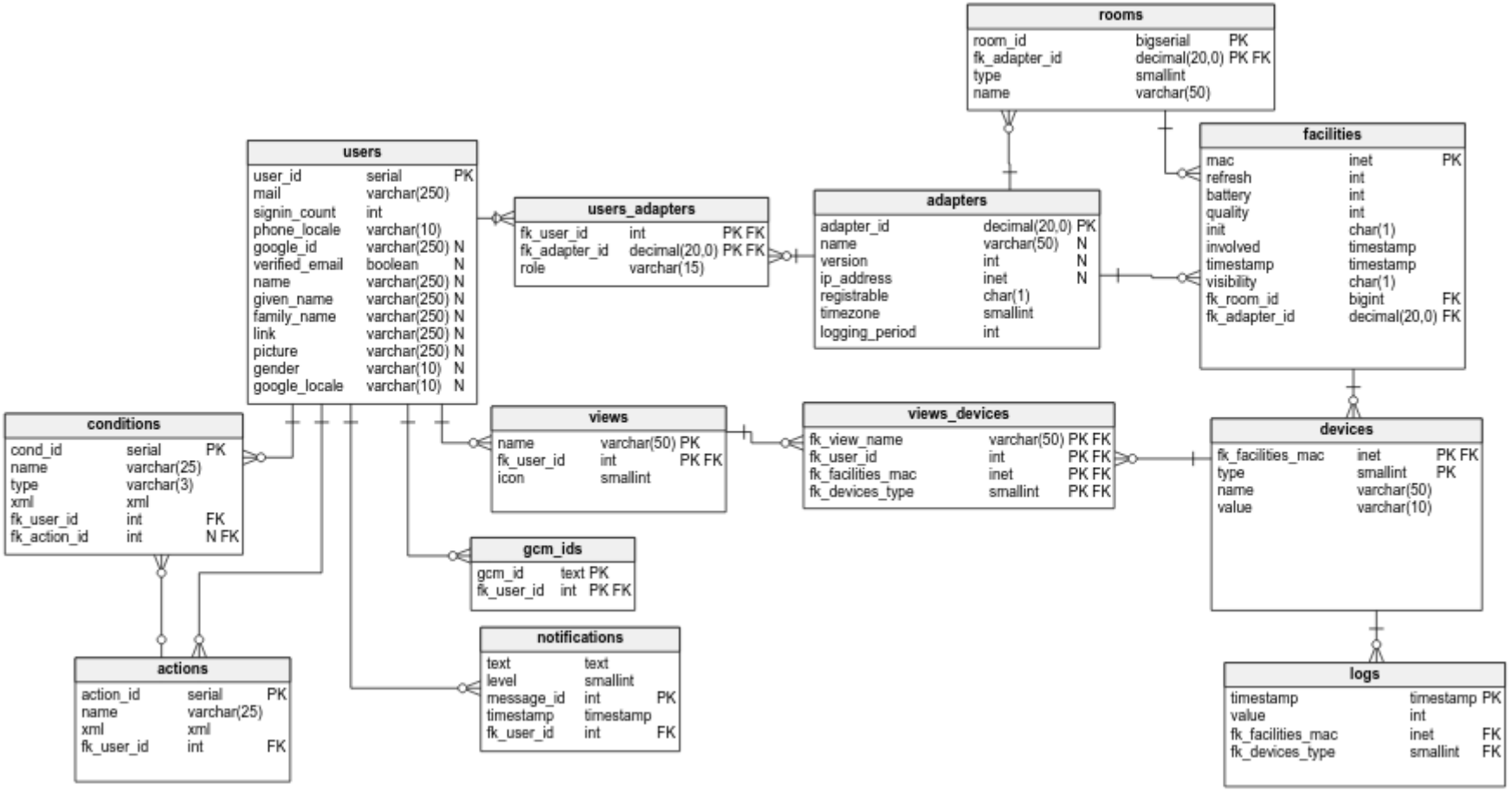
# Dodatek A

# Obsah CD

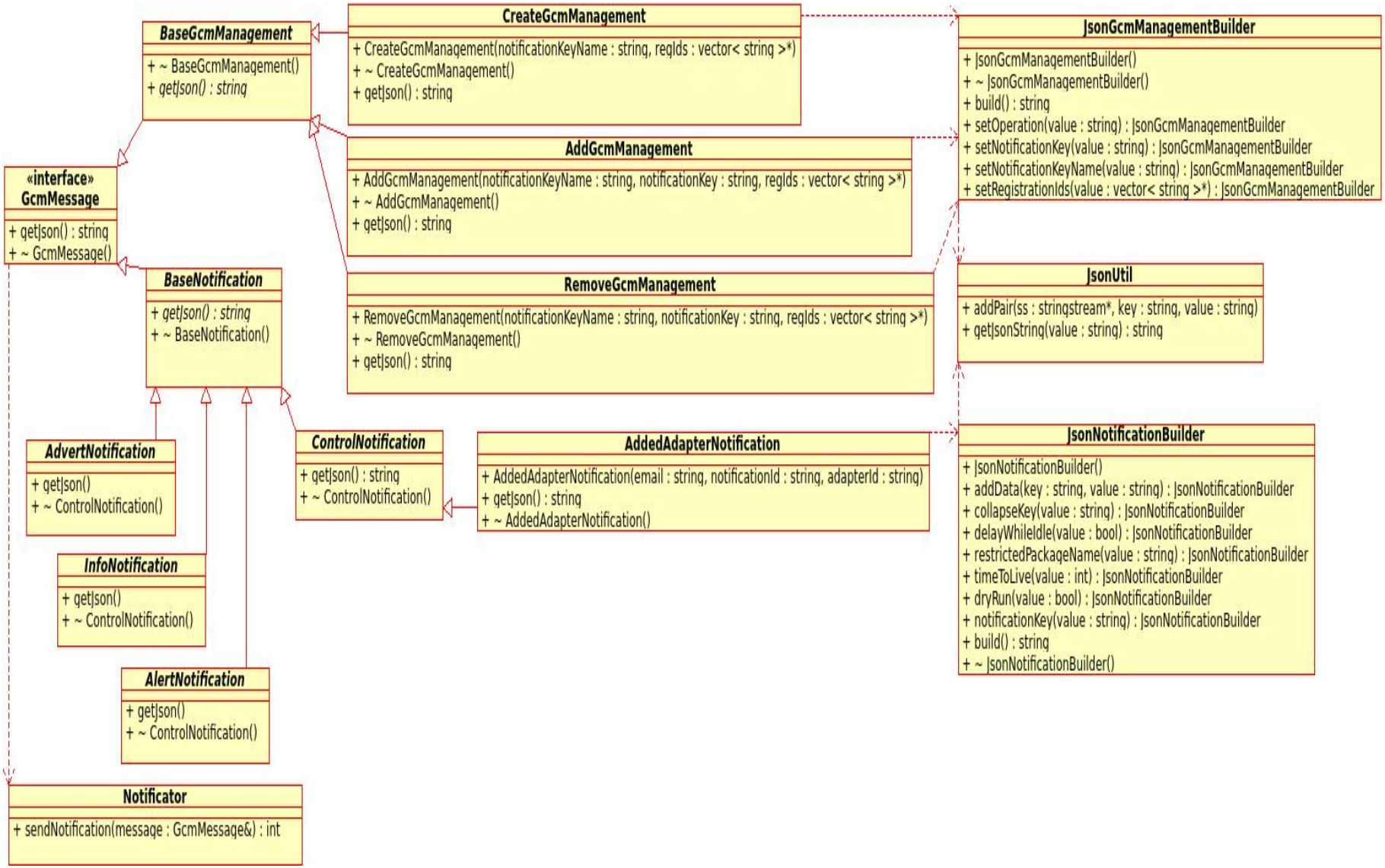
Součástí technické práce jsou také přílohy v elektronické podobě se zdrojovými soubory aplikace, které jsou umístěny na standardním nepřepisovatelném médiu. Na tomto médiu jsou tyto zdrojové soubory uloženy v adresáři `src`. Je zde také uložen soubor `readme.txt`, který popisuje v anglickém jazyce způsob překladu a spuštění projektu, a jsou zde uloženy skripty pro příkazový interpret *BASH* pro snazší použití aplikace.

# Dodatek B

# Obrázky




Obrázek B.1: Schéma databáze před vložením *modulárního prostředí*.




Obrázek B.2: Třídní diagram Notifikátoru.