Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

Platformy pro domácí automatizaci s využitím inteligentních prvků

Diplomová práce

Autor: Martin Vancl
Studijní obor: Aplikovaná informatika, AI2
Vedoucí práce: Mgr. Josef Horálek, Ph.D.

Hradec Králové srpen 2016

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 18.8.2016 Martin Vancl

Poděkování

Rád bych poděkoval vedoucímu mé bakalářské práce panu Mgr. Josefu Horálkovi Ph.D. za čas a přínosné rady k dané problematice, které mi věnoval a které mi velmi pomohly při zpracování této práce.

Anotace

Diplomová práce řeší v současné době aktuální téma domácí automatizace. Srovnává existující svobodná a komerční řešení včetně moderních vědeckých přístupů na toto téma. Protože ani jeden systém nevyhověl definovaným požadavkům, byla navržena nová inovativní architektura. Ta klade důraz na rozšířitelnost pomocí hardwaru od různých výrobců a hlavně na maximální možnou funkčnost systému. Toho dosahuje nahrazením běžně používaného modelu „centrální server – koncová zařízení“ včleněním lokálních serverů. Při výpadku části systému je jeho zbytek stále použitelný.

Abstract

Title: Platforms for home automation with intelligent features

The thesis addresses the currently hot topic of home automation. Compares an existing free and commercial solutions, including modern scientific approaches on this subject. Because neither system has not complied with defined requirements, innovative new architecture was designed. The emphasis is on extensibility by using hardware from different manufacturers, and mainly on the maximum possible functionality of the system. It accomplishes this by replacing the commonly used model of "central server – terminal equipment" by incorporating local servers. In case of failure of the system is the rest of it is still usable.

Obsah

1. Úvod

Domácí automatizace a chytré domy se těší čím dál větší popularitě. Správně fungující „chytrý dům“ může přinést jeho obyvatelům úsporu energií, ale hlavně pohodlnější bydlení.

Dům může na základě nadefinovaných pravidel vykonávat činnosti. Například: „zapni topení v pondělí v 17:00“, „pokud svítí slunce, zatáhni žaluzie“. Aby bylo možné tyto úkony provádět, je nutné mít specializovaný hardware, který zajišťuje měření dat (teplota, pohyb,…) a na druhé straně vykonávat činnosti, jako zatáhnout žaluzie, rozsvítit světla. Ať už jde o spínací reléový modul, nebo vstupní analogový teploměr, je nutné mít software, který všechna hardwarová zařízení koordinuje. Právě díky schopnostem řídícího software získá uživatel možnost definovat jednoduchá, nebo komplexní pravidla využívající data z mnoha zařízení. Ovládání je možné jak pomocí tlačítek, tak i dnes rozšířených chytrých mobilních telefonů. Pomocí software je možné vytvořit i drobnou „umělou inteligenci“.

Cílem této práce je vytvořit ovládací software pro již existující hardwarové moduly první verze projektu HAUSY. Pro zajištění maximální možné dostupnosti a robustnosti systému použijeme třívrstvou architekturu, místo běžné douvrstvé. Docílíme tak omezené funkčnosti systému i při výpadku centrálního řídícího prvku. Tato vlastnost, i když zní samozřejmě, je u komerčních řešení pro chytré domácnosti spíše vyjímečná.

Výstupem této práce bude aplikace pro „subserver“ - vrstvu komunikující s hardwarem a aplikace centrálního serveru. Centrální server bude mít REST API, pomocí kterého bude možné vytvářet další webové a mobilní aplikace. Pomocí API bude také možné nasadit jiné plánovací systémy pravidel a přizpůsobit je na míru i méně obvyklým požadavkům. Každá vrstva tak bude moct být nahrazena libovolným jiným programem implementujícím navržené API.

Tato diplomová práce navazuje na diplomovou práci Ing. Jana Štěpána: „*Návrh a implementace HW a SW smart zařízení pro komunikaci v inteligentním domě*“ [1], která se zabývá hardwarovými moduly a jejich obslužným programem pro Raspberry Pi.

2. Rešerše

Na úvod si rozebereme existující vědecké přístupy k domácí automatizaci [2]. Poté srovnáme existující „reálná řešení“ a to jak komerční, tak i svobodné.

2.1. Vědecké přístupy

Článek [3] se zabývá implementací OpenHAB v chytrých domech. Pojednává o adaptivním uživatelském rozhraní a jeho přizpůsobení uživatelským potřebám na základě připojených zařízení a definovaných pravidel [4].

V [5] se zabývají bránou pro sběr dat z automatizačních prvků v jazyku Java umožňující vytvářet simulace nad získanými daty. To je výhodné například pro testování uživatelem nadefinovaných pravidel.

Zajímavým přístupem k řešení domácí automatizace s nízkými pořizovacími náklady je například [6]. Místo serveru na řízení domu používají zařízení s mobilním operačním systémem Android a na něm vytvořenou aplikaci, které má na startosti řízení celého domu.

Ve vztahu k připojení k systému chytrého domu se zabývá článek [7]. Ten řeší různé způsoby připojení přes síť k systému, pro více typů zařízení (hardware, mobilní aplikace). Také se zabývá požadavky na zabezpečení komunikace.

Potřebu na uložení velkého množství dat v systémech domácí automatizace řeší [8]. Autoři rozebírájí a porovnávají způsoby nejefektivnějšího uložení množství naměřených dat ze senzorů a následné zpracování těchto dat.

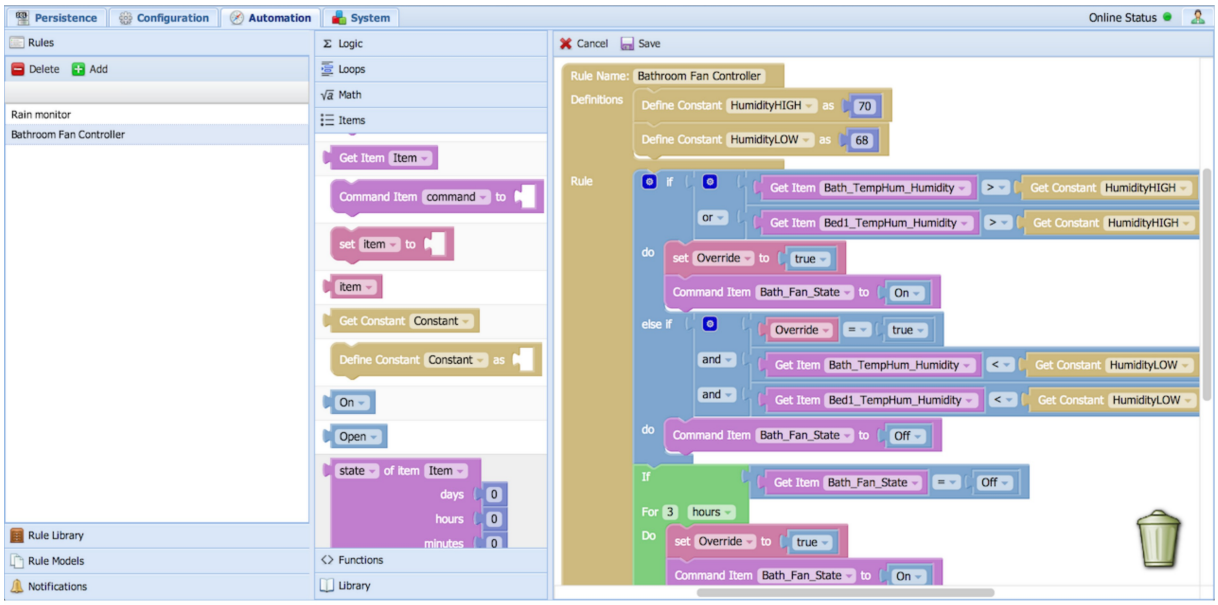
2.2. Komerční a svobodné produkty

Systémů a celkových řešení domácí automatizace existuje v současné době velké množství. Některé systémy jsou svobodné (open source [9]), jiné komerční. Svobodné systémy většinou poskytují pouze software, pomocí kterého je možné integrovat hardware od různých výrobců. Naproti tomu komerční řešení obvykle poskytují provázaný celek hardwarových modulů a ovládacího software. Na ukázku si představíme dva známé komerční a dva svobodné projekty.

2.2.1. OpenHAB

„OpenHAB je svobodný software, který slouží k řízení inteligentních domů. Sám o sobě neposkytuje přímé služby pro ovládání jednotlivých prvků (IoT), ale poskytuje jednotné prostředí pro sjednocení jejich ovládání pomocí webového rozhraní. Současně existují i aplikace pro iOS, Android a Windows Mobile, stáhnutelné z jednotlivých storů.“ [10]

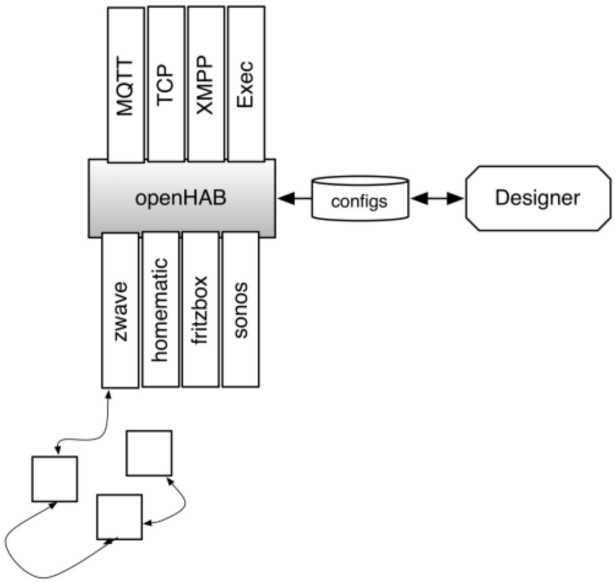
Je celý napsaný v Javě [11]; je možné ho nainstalovat na Raspberry Pi [12].



Obr. 1: OpenHAB - definování pravidel [45]

Definování pravidel je možné také pomocí „Scratch“ [13] jazyka. Uživatel může spojováním jednotlivých funkcí vytvořit algoritmus pro řízení domu. I když vypadá tento způsob programování jednoduše, ve skutečnosti v něm stejně většina uživatelů nebude schopná vytvořit složitý funkční algoritmus.

Nejdůležitější části OpenHUBu je program zvaný „runtime“. Ten funguje jako sběrnice mezi různými protokoly a sběrnicemi. Je tak možné všechna zařízení centrálně ovládat bez ohledu na jejich způsob komunikace.



Obr. 2: Architektura OpenHAB [14]

Největší výhoda OpenHABu je jeho možnost využívat hardware z různých jiných automatizačních systémů, připadně připojení k nim. Velká část těchto pluginů je však neoficiální a spravovaná komunitou. To může mít za následek nefunkčnost systému po aktualizaci využívaného hardware z jiného systému jeho výrobcem.

2.2.2. Domoticz

Je další open source automatizační systém. Umí integrovat hardware využívající rozličné komunikační technologie a sběrnice. Například Z-Wave [15], EnOcean, MySensors, 1-Wire, WiFi a spoustu dalších [16].

Domoticz [17] je na rozdíl od OpenHAB napsaný v C++, takže má menší naroky na hardware. Jako OpenHAB má webové rozhraní i aplikace pro Android a iOS (Apple).

Pokud potřebujeme velice podrobné nastavení pro velké množství uživatelů za cenu větší složitosti, má OpenHAB více možností. Domoticz má snadné webové rozhraní a síťovou konfiguraci. Primárně se hodí spíše pro použití běžným koncovým uživatelem.

2.2.3. iNELS

Je komerční systém pro domácí automatizaci původem z česka.

Firma nabízí celou řadu produktů na automatizaci domácností, ale i velkých komerčních budov. Součástí systému je jejich vlastní sběrnice „iNELS Bus system“ [18]. Pomocí té je možné připojovat nástěnné ovladače, aktory, převodníky, senzory. Většinu produktů je možné získat i v bezdrátové verzi. K ovládání slouží webové rozhraní a aplikace pro chytré telefony a tablety.

Uživatel chytrého domu vybaveného systémem iNELS může ovládat například rozsvěcení a intenzitu světel (stmívače), rolety, topení a mnohé další. Jde téměř libovolně vytvářet pravidla, takže pokud odejdou všichni obyvatelé domu, zhasnou se světla a vypne topení. Nebo se naopak světla náhodně rozsvěcují a simulují tak přítomnost obyvatel v domě. Vše záleží na nastavení.

Jak uvádí výrobce v propagačním letáku:

„Ještě než se vrátíte domů, iNELS ohřeje přichystanou večeři v troubě, zapne vyhřívání sauny a nastaví venkovní rolety podle aktuální intenzity osvětlení. Úsporný provoz regulace teploty přepne na aktivní vytápění, v letních měsících zapne klimatizaci.“ [19]

Je možné zakoupit „smart home kit“, který obsahuje vše potřebné pro nasazení. Mají sady na příklad na ovládání světel, topení, audia a videa a další.

2.2.4. Loxone

Je další z mnoha automatizačních systémů. V zásadě umí podobné věci, jako výše zmíněný iNELS. Je možné z centrálního místa ovládat světla, topení, vrata od garáže,… a definovat si vlastní pravidla v závislosti na datech ze vstupních senzorů. Pokud má uživatel ve svém domě vlastní „elektrárnu“ v podobě fotovoltaických panelů, Loxone zvládne při přebytku energie zapínat spotřebiče s velkou spotřebou a naopak při nedostatku energie a jejím nákupu z veřejné rozvodné sítě tyto spotřebiče vypnout, případně omezit jejich funkčnost.

Funkce „multiroom audio“ [20] umí přehrávat v každé místnosti jinou hudbu podle preferencí uživatele. K přehrávání je možné použít celou řadu audio prvků, nejznámější je asi Sonos. Ovládání je opět řešeno pomocí nástěnných tlačítek, webové aplikace a aplikací pro mobily a tablety.

Výrobce na webu svůj systém prezentuje tímto výstižným textem:

„Zapomeň na zastaralý systém a využij výhod inteligentní elektroinstalace. V chytrém domě od Loxone je vše řízeno centrálně a bydlení dostává zcela nový rozměr. Neuděláš už ani krok navíc. Nebudou tě zdržovat nadbytečná tlačítka pro ovládání stínící techniky, osvětlení nebo topení! Chytrý dům se ovládá sám, s pomocí malé zelené krabičky - Loxone Miniserveru.“ [21]

2.3. Problém

Problém všech uvedených komerčních řešení je jejich „vendor lock-in“ [22]. Firma se cíleně snaží o co nejmenší kompatibilitu s konkurencí. Uživatel se už při výběru systému do svého domu zaváže na mnoho let dopředu využívat produkty pouze jedné firmy. Každý systém má svoji omezenou podporu. Pokud se za deset let rozbije ovládací modul, bude se ještě vyrábět? Pravděpodobně ne. V nejhorším případě pak bude nutné udělat technické úpravy v domě a vyměnit všechny komponenty a kabeláž za nové, aktuálně prodávané.

Všechny systémy také spoléhají na centrální řídící jednotku. Při jejím výpadku je systém nefunkční, při nevhodně použité automatizaci až neobyvatelný.

Vědecké přístupy sice přináší spoustu zajímavých přístupů, ale většinou se zabývají pouze jednou malou částí problému. Aby bylo možné ovládat celý dům, je potřeba mít celkový systém – od hardwaru až po aplikaci v mobilu. Z těchto důvodů jsou vědecké práce dobrou inspirací, ale je nutné na nich dále stavět.

Řešením je použít plně svobodný systém pro domácí automatizaci, ke kterému bude bez omezení dostupná veškerá dokumentace. Dokumentaci musí exitovat jak k softwaru (open software) tak k hardwaru (open hardware). I kdyby po letech už nebylo možné koupit hardwarové moduly, nebo ne současných počítačích nebylo možné spustit řídící software, dá se problém řešit. Například si podle dokumentace nechat na zakázku vytvořit nový modul.

3. Návrh systému

Většina současných automatizačních systémů pro chytré budovy je nesvobodná a záměrně co nejméně kompatibilní s konkurenčními produkty. Uživatel si vybere systém od firmy X a poté musí kupovat snímače a akční členy [23] pouze od firmy X. Jedná se o ukázkový případ vendor lock-in. Pokud daná firma zkrachuje, nebo po deseti letech přestane starý produkt podporovat, uživatel se ocitne bez jakékoliv podpory. Jde o bezpečnostní aktualizace, nové funkce, ale také o hardware. Pokud musíme používat pouze vypínače k danému systému, tak nebudeme moci nikdy vyměnit vadné kusy, což bude znamenat kompletní výměnu automatizačního systému. Určitým řešením by mohlo být reverzní inženýrství nesvobodného protokolu. To je však často velice náročné a také v rozporu s licencí.

Proto je vhodné svobodné řešení s volně dostupnou dokumentací. I kdyby se produkt také přestal vyrábět, není problém zaplatit vývojáře, který i po letech systém upraví. Pokud navíc systém bude také open hardware, bude možné vyrábět i akční členy. A to jako náhradu za originální, ale také úplně nové.

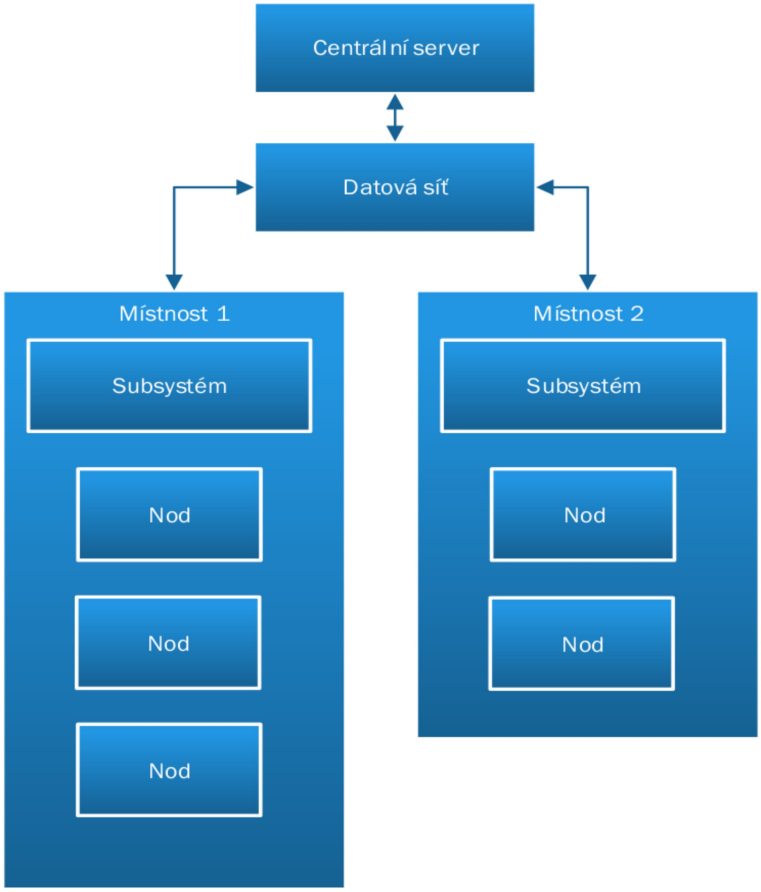
Dalším velkým problémem stavajících systémů je jejich architektura. Ta nejčastěji sestává z jedné centrální řídící jednotky a sítě senzorů a akčních členů. Jakmile dojde k poruše centrální jednotky, celý systém přestane fungovat, nebo se začne chovat náhodně – to může být ještě horší situace.

Pokud budova používá automatizační systém ke kompletnímu řízení a ne pouze jako doplněk, přestanou fungovat i základní funkce domu. Mezi ty patří například rozsvěcení světel, tekoucí voda, zásuvkové okruhy a topení.

Tyto nedostatky navržený systém řeší následovně: jde o svobodný software i hardware a místo standardní dvouvrstvé využívá novu, třívrstvou architekturu.

Systém obsahuje také centrální server „1. vrstva“, který řídí celý dům. Pomocí něho dochází k ovládání jednotlivých subsystémů, mezi než patří zásuvky, světla, topení, žaluzie a další. Uživatelova interakce s domem sestává právě z tohoto centrálního bodu. Pokud dojde k výpadku nebude možné definovat nová pravidla. Pravidla však jednou nastavíme a pak už je jenom využíváme.

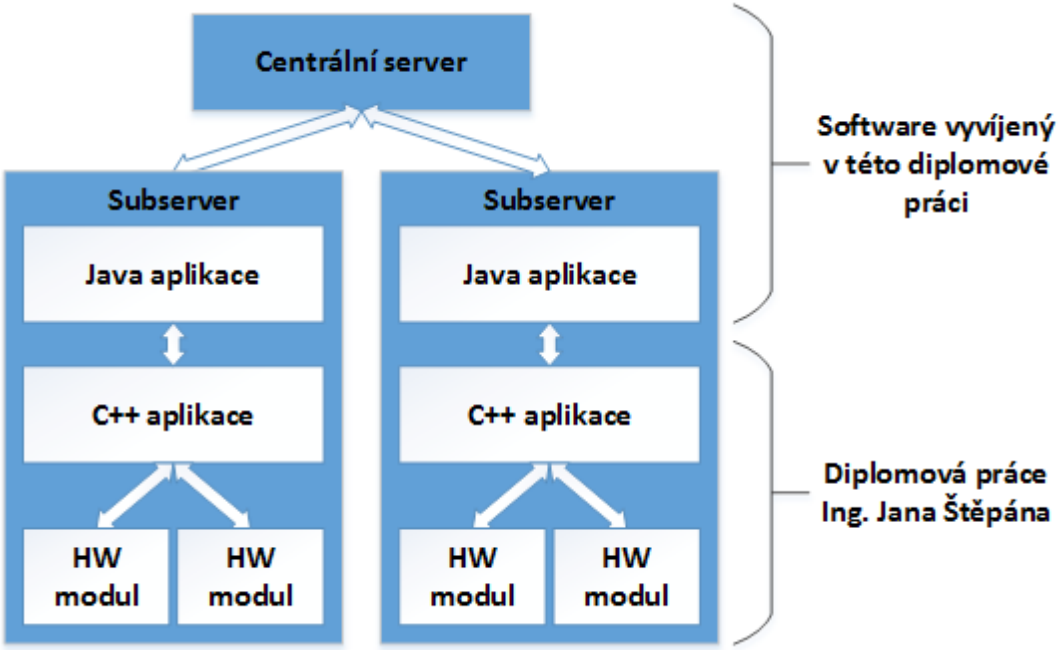
Mezi první vrstvu a koncovými zařízeními je umístěna nová mezivrstva „2. vrstva“. Té pro zjednodušení budeme říkat subserver. Subserver bude samostatný pro každou místnost, nebo logický celek. Můžeme tak mít subserver „obývací pokoj“. Bude mít na starosti



Obr. 3: Navržená architektura systému [1]

kompletní ovládání obývacího pokoje – světla, topení, žaluzie,… Také může být samostatný subserver „topení“ ovládající vytápění celého domu. Celý systém je vysoce modulární, konečné rozhodnutí závisí až na místních podmínkách. Klíčová výhoda subserveru je v jeho částečné autonomnosti. Sice nemůže definovat nová pravidla, ale pamatuje si je. Když dojde k výpadku centrálního serveru, subserver si stále pamatuje, co má dělat. Ví tak, který vypínač ovládá které světlo. Uživatel výpadek centrálního serveru nepocítí takovou měrou. Sice přestanou fungovat pravidla typu „vypínačem ve sklepě rozsviť světlo na půdě“ - tedy pravidla přes více subserverů, ale nedojde ke kompletnímu zablokování. Další důležitou funkcí je přidávání a správa koncových zařízení. Rozšíření systému o nové moduly se děje na této vrstvě. Tím myslíme například přidávání nových sběrnic (Bluetooth Low Energy [24], Ethernet, WiFi,…).

Nejnižší vrstva, „3. vrstva“ je tvořena z hardwarovách modulů. Jejich popis není předmětem této práce. Tím se zabýval kolega Ing. Jan Štěpán ve své diplomové práci [1].



Obr. 4: Schéma bloků automatizačního systému [autor]

Horní blok „centrální server“ slouží k centrálnímu řízení celého domu. Pomocí něj (API) se připojují mobilní aplikace a webové rozhraní. Shromažďuje data od všech subserverů a řídí je. Zároveň jde o jediné místo, kde je možné pracovat s pravidly. Pomocí datové sítě TCP/IP je spojen s větším množstvím subserverů.

Subserver je sada dvou aplikací provozovaných na ARMovém počítači Raspberry Pi [12]. Java aplikace přijímá/odesílá data od centrálního serveru a pomocí TCP socketu komunikuje s C++ aplikací. Udržuje si cache naměřených dat a pravidel. V případě výpadku zajistí alepsoň omezenou funkčnost systému.

C++ aplikace se stará o komunikaci s hardwarovýmo moduly, takzvanými „nody“. Sama si nedrží žádné trvalé úložiště dat. Po svém spuštění získá všechna potřebná data od Java aplikace. S hardwarovými moduly komunikuje přes sběrnici RS-485 [25].

Hardwarové moduly se starají o vyčítání stavů a spínaní akčních členů. Povely získávají přes RS-485 od C++ aplikace na subserveru.

4. Výběr hardware a software pro realizaci

Pro centrální server i subserver je nutné zvolit softwarovou platformu pro implementaci systému.

Nejdříve je nutné vybrat programovací jazyk, ve kterém budeme aplikace implementovat. Je potřeba přihlédnout k podpoře, rychlosti vývoje a učící křivce jazyka.

Obě aplikace budou uchovávat velké množství dat v relační databázi. Vybírat budeme podle rozšířenosti, rychlosti, a ceně licence. Databázový server také musí fungovat pod Linuxem.

Server i subserver vyžadují dedikované počítače, proto musíme srovnat na trhu dostupné počítače. Hlavním kritériem, obzvláště u ARM počítačů je jejich podpora v Linuxu. Ta čato bývá velmi špatná.

Operační systém bude velmi jednoduché vybrat, kvůli jednodeskovým ARM počítačům v podstatě připadá v úvahu pouze Linux a *BSD.

Aby aplikace mohly spolu komunikovat, musí být jednozačně definované API. Je možné využít „plain TCP socket“, nebo robustní protokol typu SOAP[1].

4.1. Výběr programovacího jazyka

Je důležitou části pro návrh systému. Právě on rozhoduje o portovatelnosti aplikací na různé operační systémy, rychlosti a složitosti vývoje a výkonu aplikace.

4.1.1. Nižší programovací jazky

Nižší programovací jazyky jsme hned od počátku zavrhli. Sice při správném návrhu a dobře napsaném zdrojovém kódu přinášejí nejvyšší výkon, ale mají za to velkou daň. Tou je hlavně nízká rychlost vývoje a možné bezpečnostní problémy. Často nastávají chyby jako přetečení bufferu, nebo pády celého programu při chybném, nebo chybějícím ošetření návratových hodnot všech funkcí. Při dnešním výpočetním výkonu a náročnosti námi tvořených aplikací byla výhoda rychlosti nižších programovacích jazyků zanedbatelná.

1 Simple Object Access Protocol

4.1.2. Vyšší programovací jazyky

Umožňují většinou mnohem rychlejší vývoj. Podporují výjimky a automatickou správu paměti. Celkově není tak snadné v nich napsat chybový kód jako v jazyku C.

Na výběr jsme měli C#, Python a Javu.

C# jsme museli zavrhnout hned na začátku, protože platforma .NET je určená pouze pro Windows. Sice existuje projekt Mono [26], kde se komunita snaží portovat .NET virtuální stroj na Unix-like (hlavně Linux) sytémy, ale podpora za oficiálním .NET od Microsoftu hodně zaostává.

Python je výborný pro rychlý vývoj, velké množství dostupných knihoven a má rychlou učící křivku jazyka. Na některé věci je sice pomalejší, hlavně na mnohanásobné operace s čísly. Tento probmém se však řeší využitím nativní knihovny psané v C/C++ a volané z Pythonu. Python je interpretovaný jazyk – nekompiluje se. To je další nevýhoda, Kdokoliv tak může měnit zdrojové kódy a zanést do nich chybu, nebo bezpečnostní hrozbu. Je sice možné Python aplikaci „převést“ do C/C++ a následně zkompilovat binární soubor pro danou platformu, ale není to úplně čisté řešení.

Java je jeden z nejpoužívanějších programovacích jazyků na světě. Používá se od embedded systému, typu platebních karet – Java Card, přes desktopvé aplikace – Java SE, až po weby a rozsáhlé serverové systémy, které využívají třeba banky – Java EE. Navíc je Java multiplatformní. Zdrojové kódy zkompilujeme do Java bytekódu a ten poté můžeme spustit na všech platformách s virtuálním strojem Javy.

Z výše uvedených argumentů nám vyšla nejlepší Java. Používáme verzi 1.7, ta už je dostupná ve své svobodné variantě OpenJDK v mnoha linuxových distribucích.

4.2. Výběr databáze, ORM

Celý systém potřebuje ukládat velké množství dat, takže je možné použít textové soubory. Protože používáme objektově relační mapování, rozhodli jsme se použít klasickou desetiletími ověřenou relační SQL databázi.

Rozhovali jsme se mezi následujícími databázemi: SQlite, MSSQL, Oracle a MySQL.

Embedded databáze (SQlite) jsme zavrhli. Hodí se spíše pro ukládání malého množství dat v rámci jedné aplikace.

Microsoft SQL databáze vyžaduje ke svému běhu Windows, proto ji není možné použít. Byla by tím narušena multiplatformnost celého systému.

Oracle databáze je nejznámější a v mnoha ohledech nejlepší databází. Oracle databáze jsou však velmi drahé. Cena za licenci by převýšila cenu hardware celého serveru [27].

Poslední testovanou databází byla MySQL. Je bezplatná, masově rozšířená a hlavně mnoha lety provozu ověřená. Podporuje všechny pokročilé funkce typu triggery, replikace, uložené procedury a pohledy.

Z důvodu bezpečnosti, přehlednosti, rozšiřitelnosti a udržovatelnosti nepoužíváme přímo JDBC,[2] ale objektově relační mapování. Pomocí něho pracujeme v aplikaci s běžnými Java třídami „POJO“ a ORM se stará o jejich persistenci do relační SQL databáze. Pro ORM jsme se rozhodovali mezi frameworky iBATIS, jeho forkem MyBatis a Hibernate.

MyBatis a iBATIS nemapuje Java objekty na tabulky v relační SQL databázi, ale funguje na opačném způsobu. Mapuje Java metody na SQL dotazy.

Díky tomu je možné mít velku kontrolu nad databázovými dotazy a provést jejich dokonalou optimalizaci. Pokud je hodně logiky implementováno v databázi (pohledy, uložené procedury), je *Batis vhodnou volbou.

Hibernate je ideální k persistování Java objektů. Dokáže automaticky generovat velmi komplikované SQL dotazy, jejich ruční psaní by bylo velice náročné. Díky speciálnímu dotazovacímu jazyku HQL[3] získáme vysokou nezávislost na konkrétní databázi a snadný zápis dotazů. Hibernate je jednou z nejpoužívanějších a nejrozšířěnějších implementací Java Persistence API (JPA).

Protože potřebujeme hlavně persistenci objektů a provádět hodně různých dotazů do databáze, rozhodli jsme se pro Hibernate ORM.

4.3. Výběr Java frameworku

Vaadin framework umí vytvářet takzvané RIA[4] aplikace. Ty běží v prohlížeči, ale svým vzhledem a způsobem ovládání co nejvíce připomínají klasické desktopové programy. Vývojář píše kód v Javě. Java kód je následně GWT frameworkem překládán do JavaScriptu

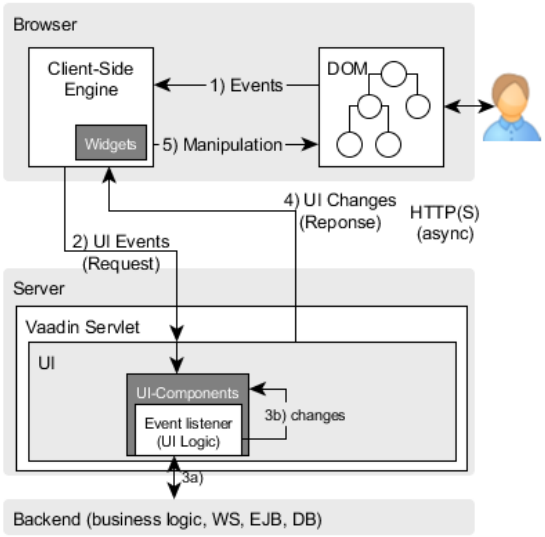
2 Java Database Connectivity

3 Hibernate Query Language

4 Rich Internet Applications

aby bylo výslednou aplikaci možné používat v běžném webovém prohlížeči bez nutnosti instalace jakýchkoliv pluginů. Všechna logika běží na serveru, klient má v prohlížeči pouze „zobrazovací část“. Vaadin je vhodný na rychlé vytvoření business aplikací s velkým množstvím formulářů a tabulek [28]. Horší je to s jeho škálovatelností. Protože na centrálním serveru máme pouze minimální administrační rozhraní, nebyl důvod použít Vaadin. Ten by šel s výhodou použít na vytvoření podrobné ovládací aplikace komunikující s centrálním serverem přes REST API [29].

JSF[5] [30] je standardní součást Jave EE stacku a tím pádem *by měla být* funkční ve všech Java EE kontejnerech. Díky knihovnám PrimeFaces a RichFaces je možné vytvářet dobře vypadající webové aplikace. JSF je ve srovnání s JSP mnohem modernější. Webová stránka se používá jako XML soubor. Jednotlivé komponenty (seznamy, tabulky) se plní daty z Javy. Hlavně kvůli příkré učící křivce jsme se rozhodli Java Server Faces nepoužít.



Obr. 5: Zpracovní událostí ve Vaadinu [31]

Spring je oblíbený open source Java framework pro vývoj Jave EE aplikací. Spring může být použit jak pro webové, tak pro klasické Java aplikace. Byl vytvořen jako alternativa k Enterprise Java Beans (EJB) a je postaven na návrhovém vzoru. Inversion of Control. Spring má výbornou podporu MVC, Hibernate ORM mapování, obsahuje součást Spring Security, pomocí které lze snadno vyřešit oprávnění uživatelů v aplikaci. Velkou výhodou je dobrá učící křivka. Díky výše popsaným důvodům a předchozím zkušenostem jsme se rozhodli k vývoji použít Spring.

5 Java Server Faces

4.4. Výběr hardware

Celý systém by měl mít co nejmenší pořizovací cenu, ale hlavně provozní cenu. Z tohoto důvodu jsme jako řídící počítače pro subservery vybírali převážně mezi jednodeskovými ARM počítači. Počítače architektury x86 jsou velké a drahé. Také jejich spotřeba je mnohonásobně vyšší, než u architektury ARM. V počáteční době projektu ještě neexistovaly jednodeskové počítače s x86, jako je nyní MinnowBoard [32].

Pro provoz centrálního serveru je možné v případě požadavků na co nejvyšší výkon a rychlost použít libovolný x86 počítač. Postačuje však výkonný jednodeskový ARM počítač.

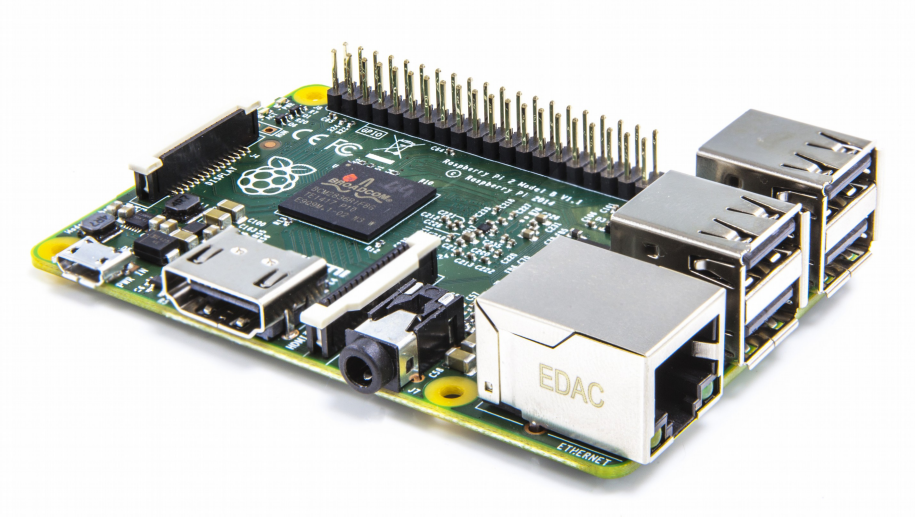
Subservery pro svoji přímou vazbu na hardware musí být na počítači se sériovým portem, USB a GPIO piny. Proto není možné použít běžné desktopové x86 základní desky. Jednodeskové ARM počítač tento počítač splňují asi všechny.

Během testování jsme postupně zkoušeli následující hardware viz Tabulka 1.

	CPU	**Jádra**	**RAM**	**Ethernet**	**WiFi**
Raspberry Pi Model B	BCM2835	1x 700MHz	256MB SDRAM	100Mbps	-
Raspberry Pi Model B+	BCM2835	1x 700MHz	256MB SDRAM	100Mbps	-
Raspberry Pi 2 Model B	BCM2836	4x 900MHz	1GB SDRAM	100Mbps	-
OrangePi PC	Cortex A7	1x 1,6GHz	1GB DDR3	100Mbps	-

Tabulka 1: Jednodeskové ARM počítače [autor]

Hlavní výhodou Raspberry je výborná komunitní podpora a podpora ovladačů. Navíc v době začátku projektu šlo o jediný reálně použitelný jednodeskový ARM počítač.



Obr. 6: Raspberry Pi 2 Model B [33]

Poslední dobou jsou velmi populární desky rodiny Orange Pi od čínské firmy Xunlong. Ve srovnání s Raspberry Pi jsou velice levné, mají vyšší výkon a některé modely i lepší hardwarovou výbavu. Problémem je softwarová podpora. Výrobce téměř neřeší ovladače a ani podporu v linuxovém jádře. Téměř o celou podporu v linuxovém jádře se v rámci komunity stará jeden šedesátiletý chorvatský inženýr vystupující na fórech pod přezdívkou „lorobos“. Orange Pi používají procesor o frekvenci 1,2GHz. Výrobce ho už z výroby má přetaktovaných na necelých 1,6GHz, což je podle datasheetu absolutní maximum. Při vyšším vytížení desky se SoC ohřeje na 80°C a začne se sám zpomalovat, aby nedošlo k jeho destrukci [34].

Ostaní desky, z nichž nejde nezmínit například Banana Pi, ODROID, Lemmon Pi, Pine64 a mnohé další mají různě výkonný hardware a příslušenství. Žádná z nich nemá 100% podporu v linuxovém jádře. Proto jsme byli nuceni zvolit bezproblémové Raspberry Pi.

Původně jsme používali, téměř ihned po jeho představení první model Raspberry Pi s jednojádrovým ARMv6 procesorem taktovaným na 700MHz a 256MB RAM. Jeho výkon a hlavně paměť nedostačovala pro běh aplikace subserveru v jazyku Java. Hlavně z důvodu, že na subserveru běží aplikace v C++, komunikující přes RS-485 sběrnici přímo s hardwarem a je závislá na vysoce přesném časování. Pomocí něho dochází třeba k přepínání směru komunikace na RS-485 sběrnici.

Raspberry Pi Model B+ s 512MB RAM vyřešilo částečně problém s nedostatkem operační paměti nutné pro běh MySQL databáze a Java aplikace. Nevyřešilo však pomalost procesoru a narušení C++ aplikace díky vysokému load average.

Nejnovější Rapsberry 2 Model B s 1GB RAM a čtyřjádrovým ARMv7 procesorem taktovaným na 900MHz vyřešilo jak problém s nedostatkem paměti, tak s výkonem. Proto jsme zvolili Raspberry Pi 2 Model B jako řídící počítač pro subservery i pro centrální server.

4.5. Výběr operačního systému

Výběr systému byl velice rychlý a jednoduchý. Nebylo totiž z čeho vybírat. V době začátku projektu a existence první verze Raspberry Pi nebylo možné spustit Windows na Raspberry. Tato možnost přišla až s nástupem Rapsberry Pi 2, kdy už byla velká část systému hotová. Navíc cena za licenci Windows by byla velkou překážkou.

Unix-like systémy z rodiny BSD jsme nepoužili z důvodu špatné podpory hardware [35] a malé komunitní podpory.

Jako jediný zbyl Linux. Je bezplatný, výborně odladěný a často na jednodeskových ARM počítačích jediným podporovaným operačním systémem (Android uvažujeme jako jednu z ditribucí GNU/Linuxu). Pro svoji stabilitu a odladěnost jsme zvolili distribuci Raspbian GNU/Linux.

4.6. Výběr komunikačních protokolů, API

Klíčovou fukcionalitou HAUSY je rozdělení celého systému na více samostatně pracujících částí. Všechny části mezi sebou komunikují pomocí definovaného aplikačního rozhraní (API). Je tak možné libovolnou část vyvinout znova podle aktuálních požadavků a zbytek systému tím nebude nijak ovlivněn.

Aby tento princip mohl fungovat, bylo nutné vybrat dostatečně robustní API. Testovali jsme XML-RPC, SOAP, REST a „plain“ TCP socket. Každá technologie má své výhody, ale i nevýhody.

4.6.1. BSD TCP socket

Je nejstarší a nejméně svazující technologií pro komunikace po síti. Poprvé byly použity v BSD Unixu verze 4.2 v roce 1983. Dá se říct, že všechny ostatní testované technologie také využívají TCP socket. Avšak nad ním definují vlastní protokol, který značně usnadňuje práci a přináší některé výhody.

U TCP socketu by bylo nutné nadefinovat vlastní formát zpráv, který by poté musely využívat všechny aplikace. Následné přidání jakékoliv dodatečné funkcionality, třeba šifrování by při počátečním nedokonalém návrhu protokolu přineslo mnoho problémů a mohl by mít za následek porušení zpětné kompatibility. Nespornou výhodou TCP socketu ve srovnání s ostatními technologiemi je jeho vysoká rychlost. Pokud však napíšeme pomalý, nevhodný parser, může být i pomalejší. Při testování na původním modelu Raspberry Pi jsme dosáhli u komunikace na localhostu propustnosti přibližně 40Gb/s.

Ze všech výše uvedených důvodů jsme návrh vlastního komunikačního protokolu nad „plain“ BSD TCP socketem zavrhli.

4.6.2. XML-RPC

Protokol XML-RPC slouží ke vzdálenému volání procedur, jak už napovídá sám jeho název RPC – Remote Procedure Call. Data jsou přenášena v XML zprávách přes HTTP protokol. Odeslání i odpověď tak obsahují klasické HTTP hlavičky.

Ve srovnání s REST, který hlavně přenáší celé dokumenty XML-RPC hlavně zajišťuje volání procedur.

Díky použití XML lze zpracovat téměř v libovolném současně používaném jazyku, avšak nám nutí použití XML. XML samo o sobě obsahuje hodně „režijních“ dat. Často jsou desítky procent přenášeného dokumentu zabrány definicí zprávy (názvy značek a jejich množství) a samotná data zabírají mnohem méně.

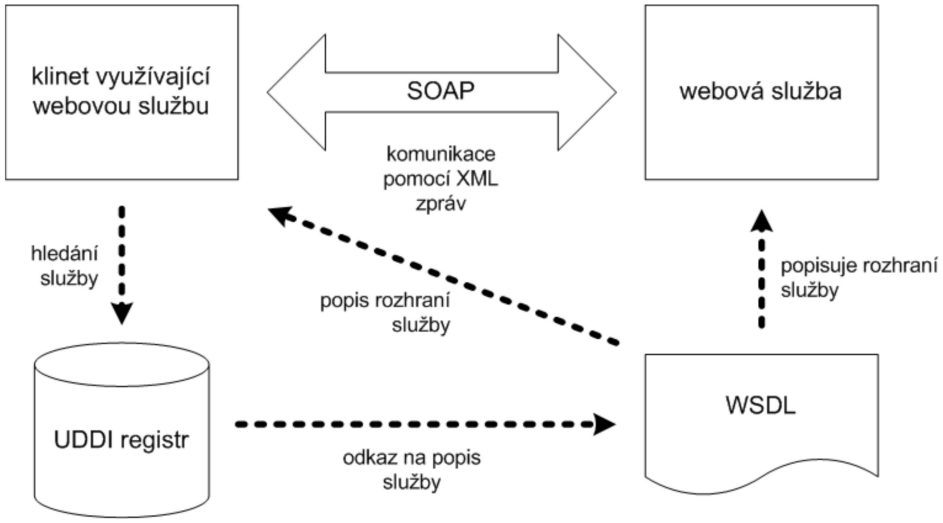
V současné době už není jeho využití v nově vznikajících projektech tolik rozšířené.

Z výše uvedených důvodů jsme neshledali XML-RPC jako nejvhodnější protokol pro naše využití v komunikaci mezi jednotlivými aplikacemi.

4.6.3. SOAP

Simple Object Access Protocol slouží k výměně dat (zpráv) nad HTTP protokolem pomocí XML zpráv. Teoreticky je možné k přenosu použít SMTP, ale reálně se využívá pouze HTTP.

Celá infrastruktura se skládá z protokolu SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language) a UDDI (Universal Description, Discovery and Integration). Všechny tři služby jsou mezi sebou provázané [36].



Obr. 7: Vztah tří technologií (SOAP, WSDL a UDDI) webových služeb [36]

Jeho nevýhodou je rozsáhlý XML zápis, který zabírá hodně místa při přenosu. XML parsování je také více náročné na výpočetní výkon. Navíc by bylo nutné využívat všechny tři výše uvedené technologie.

Proto jsme se rozhodli SOAP nepoužít.

4.6.4. REST

REST, neboli Representational State Transfer, jak zní význam této zkratky, je jak říká definice: „*architektura rozhraní, navržená pro distribuované prostředí*“ [37].

Ve srovnání s ostatními výše uvedenými protokoly (kromě plain TCP socketů) není primárně navržen na volání procedur ale na výměnu dat.

Protokol samotný v maximální možné míře využívá HTTP, vždyť také autor RESTu je jedním ze spoluautorů HTTP protokolu.

Pomocí RESTU provádíme CRUD operace (vytvoření, získání, aktualizaci a mazání) nad daty. Požadavak se definuje pomocí URL adresy. Ta má tvar například http://example.net/**api/user/1**. Ukázka definuje uživatele s ID 1. Je zvykem používat pro

CRUD adekvátní HTTP metody, tedy GET pro čtení, POST pro vytvoření, DELETE pro mazání a PUT pro aktualizaci dat. Všechny CRUD operace se většinou provádějí na stejné URL jenom s použitím odlišné HTTP metody a zaslaných dat.

REST nevynucuje použití žádného specifického formátu zprávy. Většinou se posílá JSON (JavaScrpt Object Notation), nebo XML. Nic nám ale nebrání vracet CSV, nebo spustitelné EXE soubory. Pouze nastavíme správný MIME typ.

Protože jde o protokol nad HTTP, je i jeho debugování velmi snadné. Na čtení stačí libovolný webový prohlížeč. Na ostatní metody je nutné použít specializovaný nástroj. Například terminálový curl, nebo různé pluginy do webových prohlížečů.

Podpora REST je výborná na Androidu, iOS i v JavaScriptu. Díky tomu je vývoj snadný a rychlý. Pro přidání šifrování stačí nahradit HTTP za HTTPS.

Jak vyplývá z výše uvedených odstavců, REST byl pro naše použití nejvhodnější a proto jsme ho vybrali. Pomocí něj komunikuje centrální server s uživatelským rozhraním (web, mobilní aplikace) a také se subservery.

V následující kapitole je představen podrobný systém komunikace centrálního serveru a popis definování pravidel.

5. Centrální server

Je hlavním mozkem celého systému. Ze spodní „2. vrsty“ sbírá naměřená data a naopak jim zpět posílá pravidla, podle kterých jednotlivé subservery ovládají hardware.

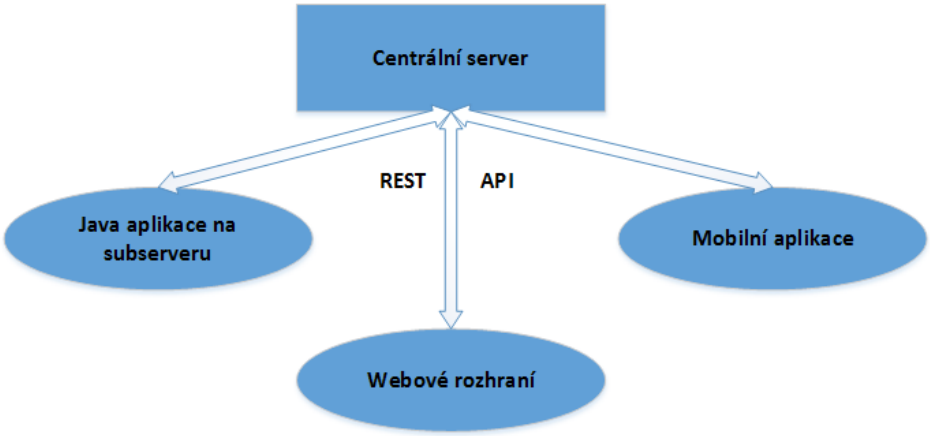
Právě tato vrstva má jako jediná v celém systému na starosti interakci s uživateli. K tomu slouží velmi jednoduché webové rozhraní určené převážně pro základní konfiguraci administrátorem systému.

Pro běžné uživatele je plánované ovládání z mobilních aplikací a nezávislého webového rozhraní, ani jedno není předmětem této diplomové práce. Oddělené uživatelské webové rozhraní má mnoho výhod, nejdůležitější jsou:

- oddělení uživatelského rozhraní od jádra systému,
- možnost změn v uživatelském rozhraní nezávisle na jádru systému,
- snížení výkonnostních požadavků – webové rozhraní může běžet na jiném serveru.

Java aplikace tvořící centrální server komunikuje s ostatními částmi systému výlučně pomocí REST API.

Mobilní aplikace i uživatelské webové rozhraní komunikuje s jádrem systému přes REST API nad protokolem HTTP 1.1. Také komunikace centrálního serveru se subservery je realizováná přes REST API. Jak subserver, tak mobilní aplikace jsou při volání RESTu vždy v roli HTTP klienta.



Obr. 8: Blokové schéma centrálního serveru [autor]

5.1. Popis činnosti

Po prvním zapnutí je databáze serveru prázdná. Server čeká na přidání subserverů pomocí administračního webového rozhraní.

5.1.1. Detekce nových nodů

Po přidání alepsoň jednoho subserveru je možné spustit detekci nových hardwarových modulů, takzvaných „nodů“ připojených k subserverům. Detekci nových nodů iniciuje vždy pouze centrální server. Hlavně z důvodu dočasné nefunkčnosti celého systému – při vyhledávní nových modulů se zastaví veškeré ostatní funkce celého systému. V tuto chvíli nefungují ani světla.

Každý nod potřebuje svoji jednoznačnou adresu, aby ho bylo možné v rámci pravidel jednoznačně identifikovat. Protože toto jediné může zaručit pouze centrální server, probíhá přídání následovně:

1. Centrální server iniciuje vyhledání nových nodů,
2. subservery si periodicky ověřují požadavek na vyhledání, pokud ho zaregistrují,
3. vyšlou pomocí TCP socketu zprávu C++ aplikaci a čekají na opověď,
4. pokud C++ aplikace vrátí nově nalezené nody, ke každému z nich subserver přidá náhodný řetězec a pošle požadavek na novou adresu centrálnímu serveru,
5. centrální server ověří existující adresy v databázi a první volnou adresu spolu s obdrženým náhodným řetězcem pošle jako odpověď zpět subserveru,
6. subserver pošle získanou adresu C++ aplikaci.

Tím je přidání nového nodu dokončeno. Náhodný řetězec se používá z důvodu současného přidání více nodů – pokud jich je nadetekováno více najednou, potřebujeme nějakou „značku“ pro spárování odpovědi na ten který dotaz.

Vrstva C++ – subserver používá náhodné číslo velikosti jednoho bajtu ke správnému spárování. Její generování obstarává C++ aplikace.

Vrstva subserver – centrální server používá vygenerovaný string, jehož generování má na starosti subserver.

Jinak je jejich význam naprosto identický, a to jak bylo zmíněno výše ke správnému spárování odpovědi na požadavek.

5.1.2. Definování pravidel

Jakmile máme alespoň jeden subserver s alespoň jedním nodem v systému, můžeme začít definovat pravidla.

Definování pravidel funguje na následujícím principu: jestliže poslední naměřená data na nodu A vyhovují definované podmínce, pak na nodu B nastav definovaná data.

Příklad: jestliže na osmikanálovém vstupním tlačítkovém nodu ovládajícím vypínače došlo ke stisknutí vypínače na třetím kanálu, nastav na osmikanálovém výstupním nodu ovládajícím světla na kanálu šest logickou jedničku.

Pravidla a nody je možné libovolně kombinovat. Všechna data jsou „pouze čísla“. Není tak problém mít podmínku ovládající žaluzie (nod motor) na základě naměřené sluneční intenzity (nod analogový vstup).

Pravidla umožňují definovat pouze jednu podmínku *if – else* a neumožňují používat logické operátory and, or a xor. K tomu slouží řetězení pravidel, jako je známé u linuxového firewallu netfilter s ovládacím nástrojem iptables. Každý nod může být použit v libovolném počtu pravidel, přičemž pravidla se vyhodnocují postupně „odshora dolů“, nebo-li „od nejstaršího k nejnovějšímu“.

Subservery si centrálního serveru periodicky dotazují na nová pravidla. Centrální server vždy na takový dotaz vrátí kompletní seznam pravidel pro daný subserver. Subserver si podle odpovědi stará pravidla smaže a nová uloží do databáze. Smysl je právě kvůli možnému výpadku centrálního serveru a následnému opakování posledních známých pravidel.

Pravidlo pracující s nody z více subserverů je pro uživatele plně transparentní. Díky abstrakci nižších vrstev nemusí řešit, ze kterých subserverů nody jsou. Pouze je použije. Takováto pravidla částečně narušují námi navržený model třívrstvé architektury. Jakékoliv pravidlo přes více subserverů se musí vyhodnocovat vždy na centrálním serveru. Problém je logicky nutný a nelze nijak vyřešit při zachování principu třívrstvé architektury.

Vyhodocování pravidel se provádí ve dvou případech. V prvním případě přišla nová data z C++ aplikace naměřená na nodech do Java aplikace subserveru. Ten si je uloží do své databáze a RESTem je přepošle okamžitě na centrální server. Pokud má nějaké pravidlo, aplikuje jej na nová data a provede jeho vyhodocení. Pokud ne, počká na možné povely definovené přes více subserverů z centrálního serveru. Druhý případ vyhodnocení je u časově definovaných akcí: „v 15:00 nastav na kanálu čtyři nodu s adresou 124 hodnotu 254“.

5.2. Databázový model

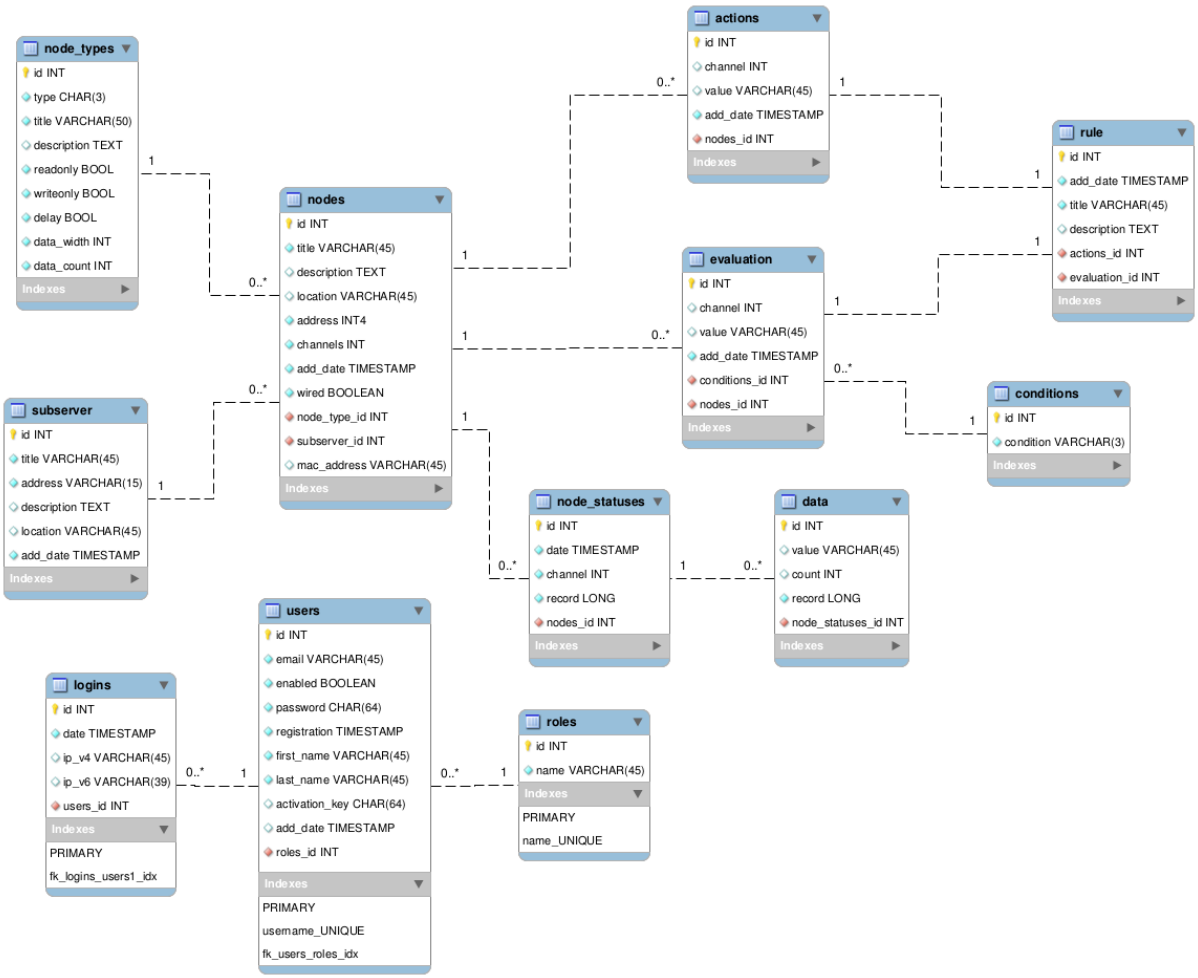
Po výběru různých databázových technologií jsme se rozhodli pro relační SQL databázi MySQL. Téměř všechna data se ukládají v databázi. Vyjímku tvoří v současné době pouze adresa subserveru, kterou si subserver ukládá lokálně v textovém souboru. Všechny nody, subservery, naměřená data i pravidla se ukládají v databázi. Uživatelská oprávnění a jejich role jsou také uložena v databázi.

Databázový model se skládá z 12 tabulek. Jejich názvy a názvy jejich atributů jsou většinou samovysvětlující.

Protože každý nod může být více kanálový, bylo nutné pro ukládání stavů nodů použít samostatnou tabulku data s vazbou na na tabulky *node_statuses*.

Podmínky řešíme porovnáváním pomocí základních operátorů <, >, =, <=, >=. Pravidel může být více a zpracovávají se postupně popořadě. Pravidla se skládají z tabulek *nodes – evaluation – rule – actions – conditions*. Pokud je nad daty nodu X splněna podmínka (*evaluation*, *conditions*), tak je vykonána nad nodem Y akce (*actions*). Pravidla je možné definovat mezi různými nody a také mezi různými subsystémy.

Pro možnosti statistického zpracování logujeme data všech událostí, přidání nodů, jejich stavů, přihlášení uživatelů a dalších.



Obr. 9: Databázový model – tabulky [autor]

5.3. Instalace a uvedení do provozu

Centrální server HAUSY je schopen běžet na počítači s Linuxem obsahujícím běhové prostředí Javy 1.7, databázi MySQL a servletový kontejner Apache Tomcat. Databázi je možné mít na samostatném stroji, ale z důvodu rychlosti a zamezení možným výpadkům je lepší provozovat databázi na stejném stroji, jako centrální server.

I když aplikace funguje se „svobodnou“ Javou OpenJDK, doporučujeme použít originální Javu od firmy Oracle a to v nejnovější verzi [38].

Operační systém může být libovolná Linuxová distribuce. Testování probíhalo na *Ubuntu Server 14.04 x86_64* a *Raspbian Jessie* pro Raspberry Pi. Na ostatních distribucích nemůžeme garantovat plnou funkčnost systému.

Pro testování jsme používali notebook s Intel Core i5 procesorem a počítač s Intel Atom N270 procesorem. Z jednodeskových počítačů pak Raspberry Pi 1 a 2. Na Core i5 a Raspberry Pi 2 funguje systém bez problému. Raspberry Pi 1 a Atom N270 svým výkonem nedostačují plynulému chodu celého systému. Pokud nám nevadí zhoršená reakční doba, je možné používat i Raspberry Pi první generace. Důrazně však doporučujeme použít Raspberry Pi 2, nebo 3, případně x86 procesor typu Intel Pentium a lepší.

Dále budeme předpokládat distribuci Raspbian Jessie na Raspberry Pi 2.

Instalaci Raspbianu provedeme jeho zapsáním na SD kartu o velikosti 8GB, případně větší. Ihned poté je dobré expandovat souborový systém do zbytku volného místa SD karty.

Zapsání obrazu a následné zvětšení oddílu provedeme příkazy:

```
lsblk
dd if=raspbian.img | pv | dd of=/dev/sdb bs=4k
umount /dev/sdb1 && umount /dev/sdb2
echo ", +" | sfdisk -N 2 /dev/sdb
umount /dev/sdb2
e2fsck -f /dev/sdb2
resize2fs /dev/sdb2
sync
```

Prvním příkazem si zobrazíme výpis všech disků a oddílů připojených k počítači. Je důležité použít správné označení disku, jinak dojde k nenávratné ztrátě dat. Na ukázce používáme pro SD kartu označení /dev/sdb. Druhým příkazem (dd) zapíšeme obraz, příkaz pv zajistí zobrazení aktuálního průběhu při zápisu. Zbývající příkazy zajistí fyzické zvětšení oddílu a roztažení ext4 filesystému do zbytku oddílu. Příkazy umont jsou nezbytné, protože není možné upravovat ext4 filesystém, který je připojený.

Po prvním nabootování je potřeba aktualizovat balíčky a poté nainstalovat MySQL databázi, Oracle Javu a Apache Tomcat. Vše provedeme jako root pomocí následujících příkazů:

```
apt-get update
apt-get dist-upgrade
apt-get install oracle-java7-jdk tomcat7 tomcat7-admin tomcat7-user \
 mysql-server phpmyadmin
```

Samotný deploy hausy serveru provedeme nakopírováním souboru *hausy_central_server.war* do adresáře */var/lib/tomcat7/webapps*. [39]

Dále je potřeba naplnit databázi ze souboru *hausy_central_server.sql*. K tomu s výhodou můžeme využít phpMyAdmin, který běží na adrese serveru *http://hostname/phpmyadmin*. Nastavení přístupů do databáze nalezneme v souboru */var/lib/tomcat7/webapps/hausy-main-server/WEB-INF/classes/jdbc.properties*, který má následující samovysvětlující syntaxi:

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.hibernate.dialect=org.hibernate.dialect.MySQLInnoDBDialect
jdbc.url=jdbc:mysql://localhost:3306/hausy
jdbc.username=hausy
jdbc.password=hausy
```

Po naplnění databáze restartujeme Tomcat:

```
service tomcat7 restart
```

V této chvíli je centrální server plně načtený a můžeme ho ovládat z webového prohlížeče na adrese *http://localhost:8080/hausy/*, kde localhost nahradíme hostname serveru.

Na této adrese je velice jednoduché webové rozhraní sloužící pouze k základním úkonům. Běžný uživatel s ním ani nemusí přijít do styku. Na ovládání totiž bude sloužit mobilní aplikace a pokročilá webová aplikace – obě budou s centrálním serverem komunikovat pomocí REST API. Webové rozhraní centrálního serveru tak využijeme pouze v případě problému s pokročilým uživatelským rozhraním, nebo při prvotní instalaci.

6. Lokální server – „subserver“

Lokální server je unikátním prvkem HAUSY. Slouží pro přímou komunikaci s připojenými hardwarovými moduly a centrálním serverem. Jeho úkolem je předávání dat mezi těmito dvěma vrstvami a v případě výpadku centrálního serveru, nebo spojení s centrálním serverem ho zastoupit do jeho opravy. To znamená, že subserver si pravidelně stahuje pomocí REST API pravidla z centrálního serveru, která se ho týkají. Samozřejmě fungují pouze pravidla v rámci daného subsystému. Pravidla „napříč“ několika subsystémy z principu fungovat nemohou.

Velkou výhodu subserveru je také možnost připojit lokální ovládací prvky, například dotykový LCD display.

Hardware a systémové požadavky jsou v mnohém společné se subserverem, až na to, že zde je v aktuální verzi potřeba použít Raspberry Pi. Opět doporučujeme model 2 nebo 3, a to hlavně z důvodu MySQL databáze a Javy.

6.1. Popis činnosti

Subserver má na starosti následující úkony:

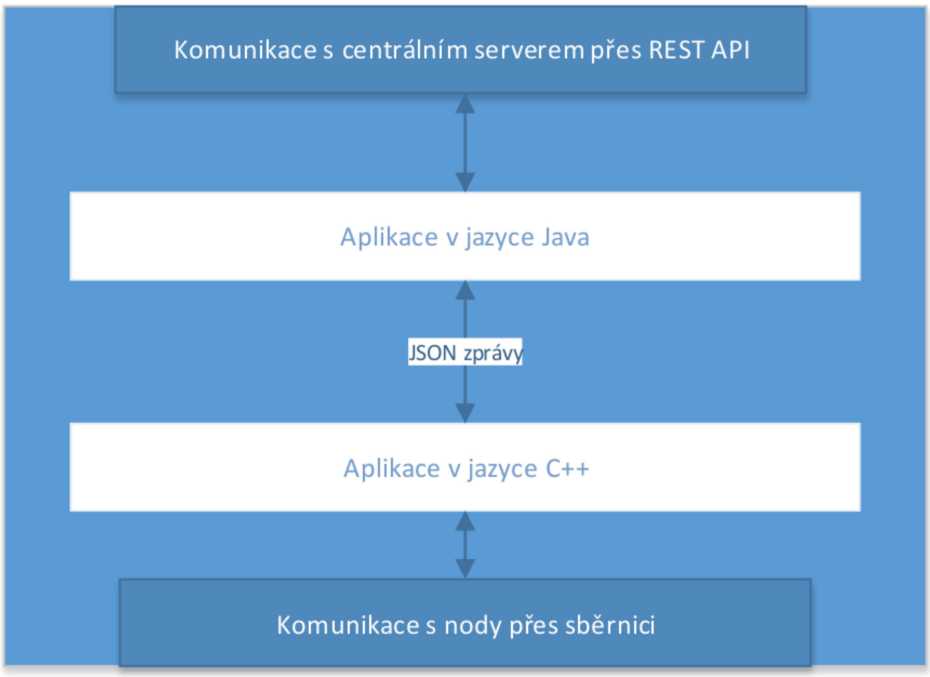
- vyčítání dat z připojených nodů a jejích zasílání na centrální server,
- po obdržení pokynů od centrálního serveru poslat nastavení nodům,
- po vyžádání detekci nodů (vždy iniciuje centrální server) zahájit detekci,
- při výpadku centrálního serveru rozhodovat dle posledních obdržených pravidel,
- po obnovení spojení poslat na centrální server všechna naměřená data.

Pro detekci nových nodů se subserver každých 10 sekund pomocí REST API dotazuje centrálního serveru na adrese *http://localhost:8080/hausy/api/v1/nodes/discovery* kde localhost je adresa serveru. Na této adrese subserver obdrží jednu z následujících JSON zpráv:

```
{"discovery":false,"timestamp":0}
{"discovery":true,"timestamp":1467672622377}
```

První znamená nevyhledávat nové nody, druhý vyhledávat. Timestamp slouží k „id“ požadavku na vyhledávání. Centrální server drží hodnotu *discovery=true* po dobu 30 sekund a některé subservery by tak mohly několikrát spustit vyhledávání. Proto si subserver po zahájení vyhledávání uloží timestamp ze serveru a při dalším pokusu o vyhledávání ho porovná. Pokud je je rozdílný o 30 sekund a více, spustí nové vyhledávání.

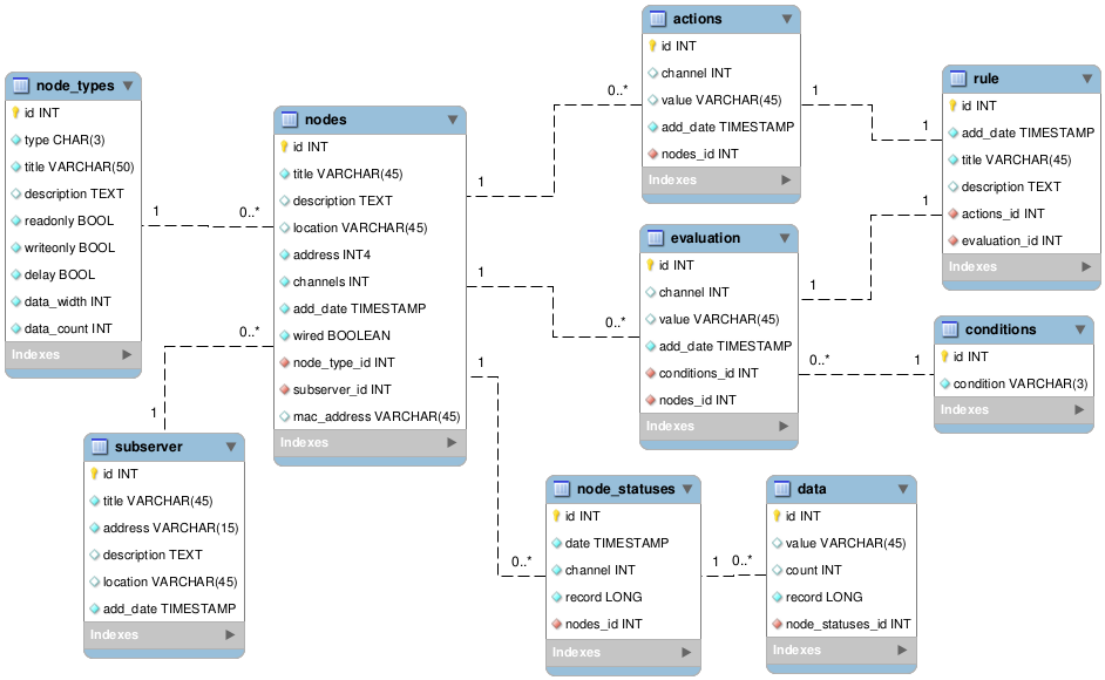
Na subserveru není možné definovat pravidla. Ta se přejímají z centrálního serveru.



Obr. 10: Komunikace mezi vrstvami [1]

Na Obr. 10 je schéma softwaru a jeho komunikace na subsystému – Raspberry Pi.

6.2. Databázový model



Obr. 11: Databázový model – tabulky [autor]

Databázový model je hodně podobný modelu centrálního serveru. Hlavní rozdíl je v absenci tabulek pro definici uživatelů. Tabulka subserver obsahuje pouze sebe sama – aktuální subserver. Taktéž tabulky s pravidly mohou obsahovat pouze poslední pravidla. To záleží na centrálním serveru. Ten může subserveru zaslat pouze poslední pravidla, nebo více pravidel.

6.3. Instalace a uvedení do provozu

Je opět hodně podobná centrálnímu serveru. Použijeme stejný Raspbian Jessie, ale neinstalujeme Tomcat. Můžeme také vynechat phpMyAdmin a tím zabránit instalaci a běhu Apache webserveru. V tomto případě bychom ale museli používat konzolového MySQL klienta. Ve většině případů instalace phpMyAdmina vůbec nevadí.

Vše provedeme jako root pomocí následujících příkazů:

```
apt-get update
apt-get dist-upgrade
apt-get install oracle-java7-jdk mysql-server
```

Ke spuštění stačí příkaz:

```
java -jar subserver.jar
```

Protože standardně není MySQL databáze na subserveru dostupná pro další počítače na síti, používá subserver vždy jméno, heslo a název databáze „hausy“. Pokud bychom z bezpečnostních důvodů chtěli přístupové údaje změnit, musíme v projektu subserveru upravit soubor src/main/resources/jdbc.properties:

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
jdbc.url=jdbc:mysql://localhost:3306/hausy-subserver
jdbc.username=hausy-subserver
jdbc.password=hausy-subserver
```

a poté znovu sestavit JAR soubor. K tomu potřebujeme mít nainstalovaný *maven*. Pro sestavení spustíme v adresáři projektu:

```
mvn package
java -jar target/subserver-java-0.0.1-SNAPSHOT.jar
```

výsledný soubor je plně funkčním subserverem.

Do MySQL databáze nahrajeme soubor *subserver.sql*. Pokud změníme přístupové údaje, je nutné znovu sestavit Java aplikaci viz výše.

Adresa subserveru je uložena v textovém souboru */root/subserver_addr.txt* a obsahuje pouze číslo adresy, například:

```
25
```

Poslední údaj je adresa centrálního serveru. Ta může být ve tvaru IP adresy, nebo doménového jména a je uložena v souboru */root/central_server.txt*. Soubor opět obsahuje jenom adresu, například:

```
10.123.10.100
```

Po nastavení spustíme subserver a tím je nastavení dokončeno:

```
java -jar subserver.jar
```

a systém je připraven na prvotní detekci nodů iniciovanou subserverem.

7. Realizované řešení komunikace s API

V této kapitole si popíšeme komunikační protokol centrálního serveru a subserveru. Z velké části jde o REST API, pouze Java aplikace na subserveru komunikuje s C++ aplikací na subserveru pomocí TCP socketu.

Důležitý je hlavně formát zpráv, které si aplikace mezi sebou zasílají.

7.1. Komunikační protokol mezi subserverem a C++ aplikací

Jak jsme si již popsali v minulých kapitolách, subserver komunikuje s nižší vrstvou přes TCP socket s využitím JSON [40] zpráv.

JSON[6] byl zvolen pro své široké použití a hlavně možnost snadného debugování. Pokud bychom použili binární protokol, jakým je například Google Protocol Buffers, nebo Google Flat Buffers, ztížilo by se ladění. U textového protokolu ihned bez jakýchkoliv speciálních nástrojů ihned vidíme a obsah zpráv. Protože používáme „raw“ TCP socket a obě části běží na localhostu, není problém posílat zprávy s větší velikostí.

Komunikaci inicují obě strany, každá v jiném případu.

Software subserver, psaný v Javě vystupuje jako TCP server, ke kterému se připojuje – jako klient C++ aplikace.

Všechny uvedené zprávy jsou validní podle RFC 7159 [41]. Všechny atributy u daných zpráv jsou povinné a není možné je vynechat. Pokud z nějakého důvodu data poslat nemůžeme, například jde o nově přidaný nod, který ještě nemá data, pošleme prázdné pole/záznam. Vždy je to však explicitně uvedeno.

7.1.1. Postup komunikace po spuštění

Po spuštění subserver čeká na příchozí spojení od C++ vrstvy. Jakmile k němu dojde, C++ vrstva ihned pošle požadavek o sdělení adresy subserveru:

```
{
 "message_type":1
}
```

Na tuto zpávu subserver odpovídá:

6 JavaScript Object Notation

```
{
 "address":123,
 "message_type":2
}
```

kde **123** je adresa subserveru datového typu **bajt**. Ihned po získání adresy si C++ apliakce vyžádá seznam typů všech nodů:

```
{
 "message_type":12
}
```

odpověď na je JSON pole:

```
{
 "types":[
 {
 "read_write":0,
 "latency":0,
 "data_range":1,
 "type":1,
 "data_count":1
 } , ...
 ],
 "message_type":13
}
```

pole **types** obsahuje **n** záznamů všech nodů. Atributy mají následující význam:

- **read_write** – určuje, zda je nod pro čtení (0), pro zápis (1), nebo pro čtení i zápis (2),
- **latency** – okamžité (tlačítko), nebo zpožděné čtení (senzor s dlouho trvajícím měřením),
- **dat_range** – rozsah dat,
- **data_count** – počet dat,
- **message_type** je identifikace zprávy.

Poslední žádost od C++ vrstvy je na seznam připojených nodů. C++ totiž nemá žádné persistentní úložiště, a proto si veškerá data uchovává v proměnných v RAM, případně v databázi subserveru.

```
{
 "message_type":8
}
```

Odpovědí je JSON pole. To je buď prázdné, pokud neexistují žádné nody:

```
{
 "nodes":[ ],
 "message_type":9
}
```

nebo obsahuje jejich seznam:

```
{
 "nodes":[
 {
 "channel_count":8,
 "address":123,
 "data":[
 {
 "data":[
 0
 ],
 "channel":0
 },
 {
 "data":[
 0,
 255
 ],
 "channel":1
 }
 ],
 "type":1
 }
```

```
 ],
 "message_type":9
}
```

Pole nodes obsahuje záznam pro jednotlivé nody. Každý nod má svojí adresu a **address** a počet kanálů **channel_count**. Dále obsahuje každý nod pole dat **data**. Pokud data zatím neexistují, například u nově přidaného nodu, vložíme prázdné pole.

7.1.2. Autodetekce nově přidaných nodů

Aby mohly nody na sběrnici komunikovat, musejí mít adresy. Každý z nich musí mít v rámci sběrnice svoji unikátní adresu. Pokud by dva nody měly adresu stejnou, docházelo by k neustálé kolizi. Nově přidané nody nemají žádnou adresu a jsou v discovery módu – čekají na požadavek k adresaci. Adresace se vždy spouští ručně na pokyn uživatele prostřednictvím centrálního serveru. Při vyhledávání dojde na cca půl minuty k vypnutí funkce všech nodů připojených ke všem subserverům, proto je autodetekce vždy spouštěna pouze na pokyn uživatele.

Průběh detekce spočívá v několika krocích. Uživatel nejdříve na centrálním serveru zadá pokyn k autodetekci. Ihned poté server na URL

```
/hausy/api/v1/nodes/discovery [HTTP GET, mime="application/json"]
```

nastaví atribut **discovery** na **true** a přidá aktuální timestamp:

```
{"discovery":true,"timestamp":1457471908707}
```

Po dobu třiceti sekund je hodnota **true**. Poté se změní zpět na **false** a nastaví se aktuální timestamp.

Všechny subservery se každých pět sekund dotazují na URL

```
 /hausy/api/v1/nodes/discovery [HTTP GET, mime="application/json"]
```

Pokud je atribut na hodnotě **true** a zároveň **timestamp** je minimálně o 30 sekund starší, než jejich lokálně uložený, spustí detekci. Ta sestává z odeslání požadavku do socketu C++ aplikaci:

```
{
 "message_type":15
}
```

Po jeho přijetí začne C++ aplikace detekovat nové nody, což trvá přibližně 30 sekund. Zpátky se v případě nenalezených nodů vrátí prázdné pole:

```
{
 "nodes":[],
 "message_type":3
}
```

Pokud byly nějaké nody nalezeny, obsahuje pole následující atributy:

```
{
 "nodes":[
 {
 "type":7,
 "channel_count":1,
 "random":1598763
 }, ...
 ],
 "message_type":3
}
```

Type a **channel_count** jsou samovysvětlující, proto se jimi nebudeme zabývat. Pro adresaci je důležitá hodnota **random**. Tu si každý nod vygeneruje a pošle spolu s požadavkem o adresu. Subserver naparsuje příchozí JSON a sestaví z něho zprávu pro centrální server následujícího formátu:

```
{
 "newNodeAddress":"033533208f0d167031cc11af4fc0307b",
 "title":"AUTO_ADD",
 "address":999,
 "channels":1,
 "nodeType":{
 "type":"7"
 },
 "subserver":{
 "address":"123"
 }
}
```

Pošle ji na URL

```
/hausy/api/v1/nodes/discovery [HTTP POST, mime="application/json"]
```

Jak je vidět, i v případě více nově přidávaných nodů se musí posílat požadavky jednotlivě a ne v poli dohromady. Atributy znamenají:

- newNodeAddress – náhodný identifikátor,
- title – jméno nově přidávaného nodu,
- address – dočasná adresa, její hodnota je v této zprávě libovolná, doporučujeme 999,
- channels – počet kanálů nově přidávaného nodu,
- nodyType{type} – typ nodu,
- subserver{address} – adresa subserveru.

Náhodný identifikátor má stejný význam, jako random v předešlé zprávě. Slouží ke spárování požadavku a odpovědi v případě několika současných požadavků na adresaci. Uplatňujeme podobný princip, jako u NAT tabulky.

Centrální server vybere první volné ID z databáze a pošle ho zpět se stavovým kodém HTTP 201[7].

Subserver použije obdrženou adresu a vloží ji do zprávy s **message_type 4**:

```
{
 "nodes":[
 {
 "address":36,
 "type":7,
 "channel_count":1,
 "random":1598763
 }, ...
 ],
 "message_type":4
}
```

random slouží pro spárování odpovědi a **address** je nová unikátní adresa.

7 201 Created

7.1.3. Odebrání nodů

Odebrání nodů je logickým opakem k jejich přidávání. Může nastat z mnoha důvodů – porucha, přemístění do jiného subserveru, prodej,…

K odebrání pošleme ze subserveru zpávu **11** spolu s adresou odebíraného nodu:

```
{
 "address":123,
 "message_type":11
}
```

7.1.4. Získávání dat z nodů

Vstupní nody neustále měří data svých vstupních pinech a senzorech. C++ aplikace je z nich periodicky po sběrnici RS-485 vyčítá a posílá je subserveru. Vždy se jedná o samostatné JSONy pro každý nod. Ukázka pro 1 kanálový nod:

```
{
 "address":123,
 "data":[
 {
 "data":[
 1
 ],
 "channel":0
 }, ...
 ],
 "message_type":5
}
```

Pro vícekanálové nody je více záznamů v sekci data na nulté úrovni. Subserver po přijetí a naparsování zprávy data v nezměněném formátu přepošle na:

```
/hausy/api/v1/nodeStates [HTTP POST, mime="application/json"]
```

o úspěšném přidání je informován HTTP stavovým kódem 201.

Zároveň s odesláním si subserver uloží data do své lokální databáze. Má k tomu dva důvody. Pokud by se nepodařilo nové stavy odeslat na server, bude se o to neustále pokoušet znovu. Druhý důvod je kvůli pravidlům. Pravidlo vždy vyhodnocuje podmínku vůči posledním datům. Pokud by došlo k přerušení spojení s centrálním serverem, přestalo by zároveň s tím fungovat vyhodnocování pravidel.

7.1.5. Nastavení stavu nodů

Zatímco do nodů nejde (a ani nesmíme) zapisovat, výstupní nody zápis ke své funkci vyžadují. Jde například o vypínače světel. Ty podle pravidel spínají relé/stykače s připojenými světly. Nový stav subserver posílá C++ vrstě a to v libovolnou dobu. Iniciátor komunikace pro nastavení světel je subserver. Zpráva se liší podle typu nodu. Pro n-kanálový vypínač:

```
{
 "address":123,
 "data":[
 {
 "data":[
 0
 ],
 "channel":0
 }, ...
 ],
 "message_type":6
}
```

obsahuje ve vnořené sekci data vždy jednu hodnotu 0/1 (on/off). Jiné nody mohou vyžadovat více dat. Jde třeba o RGB žárovky. Zde posíláme hodnoty tři pro každou barevnou složku (R, G, B):

```
{
 "address":123,
 "data":[
 {
 "data":[
 0,
 255,
 0
```

```
 ],
 "channel":8
 }, ...
 ],
 "message_type":6
}
```

na kanálu 8 bude svítit zelené[8] světlo.

8 R=0, G=255, B=0

7.2. REST API centrálního serveru

Hausy je od počátku navrhované co nejvíce modulární. Každou část je možné v případě potřeby nahradit jinou, nebo používat více implementací současně. Aby to bylo možné, je potřeba využívat ve všech aplikacích otevřené a dobře zdokumentované API. Všechny zprávy se posílají ve formátu JSON. Vždy, pokud není explicitně uvedeno jinak, vyžadujeme JSON nastavený v HTTP hlavičkách:

```
Content-Type: application/json
Accept: application/json
```

Díky tomu je možné velice snadno psát nové moduly a provádět debugování.

V současné verzi používáme protokol HTTP 1.1 bez šifrování. Pokud by bylo potřeba, stačí na všech místech zaměnit v URL protokol HTTP za šifrovaný HTTP**S**.

7.2.1. Ukázka debugování pomocí CURL

Díky REST API používajícímu pro výměnu zpráv textový JSON můžeme velmi snadno debugovat obsah zpráv. GET požadavek můžeme simulovat programem wget, nebo libovolným webovým prohlížečem. Na ostatní HTTP metody (POST, DELETE,…) je asi nejlepší program cURL [42].

Níže je ukázka HTTP GET na zjišťování požadavku na detekci:

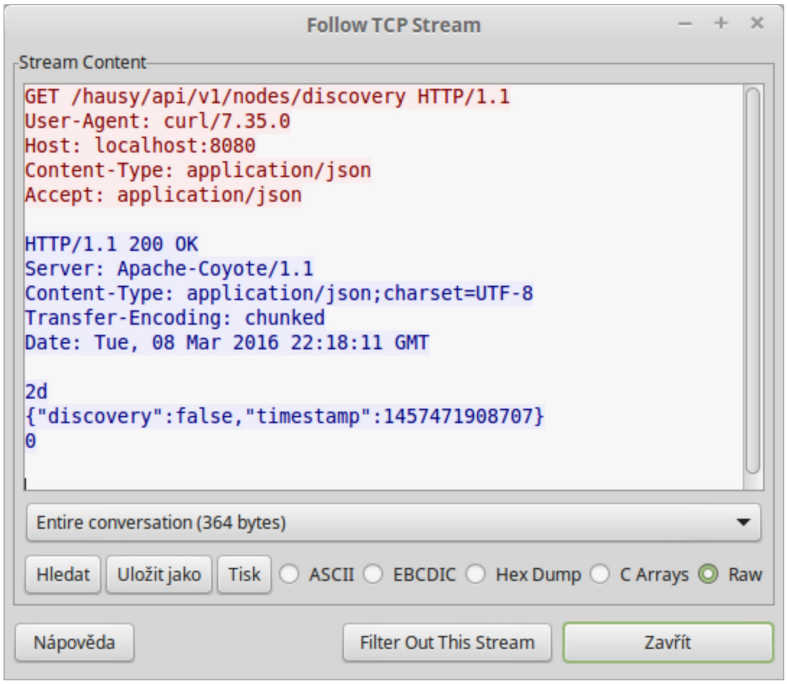
```
$ curl -X GET -H "Content-Type: application/json" -H "Accept: application/json"
http://localhost:8080/hausy/api/v1/nodes/discovery

{"discovery":false,"timestamp":1457471908707}
```

Pro účely ladění je často dobré vidět kompletní HTTP hlavičky. K tomu slouží přepínač **-v**:

```
$ curl -v -X GET -H "Content-Type: application/json" -H "Accept:
application/json" http://localhost:8080/hausy/api/v1/nodes/discovery

* Hostname was NOT found in DNS cache

* Trying 127.0.0.1...

* Connected to localhost (127.0.0.1) port 8080 (#0)

> GET /hausy/api/v1/nodes/discovery HTTP/1.1

> User-Agent: curl/7.35.0

> Host: localhost:8080

> Content-Type: application/json
```

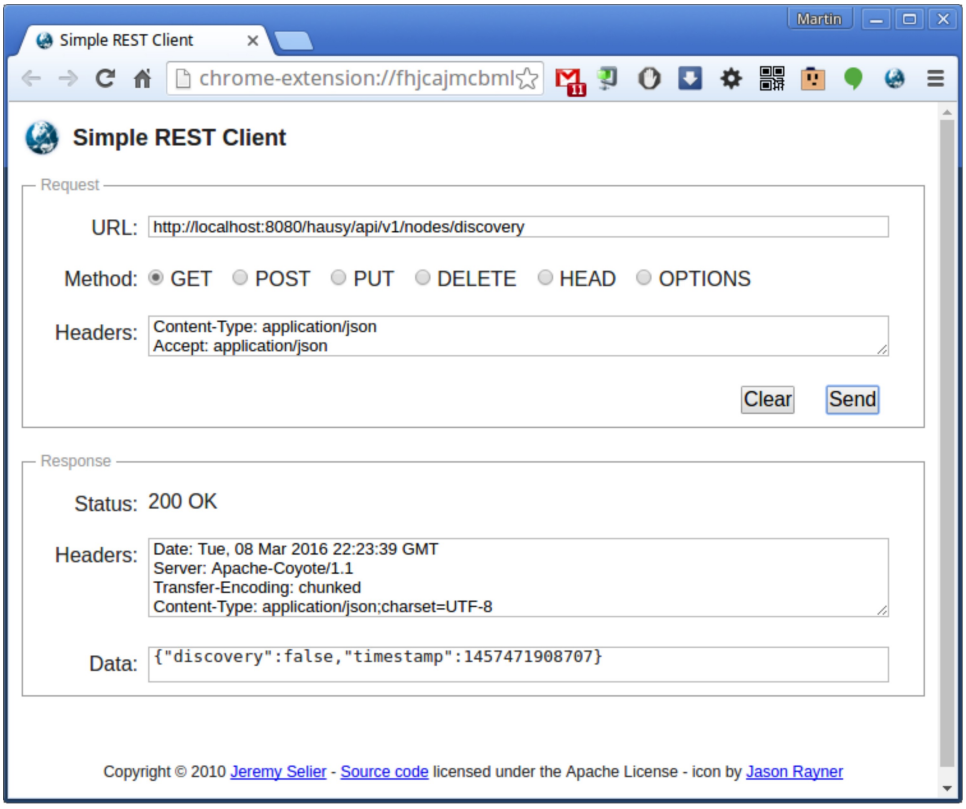
```
> Accept: application/json
>
< HTTP/1.1 200 OK
* Server Apache-Coyote/1.1 is not blacklisted
< Server: Apache-Coyote/1.1
< Content-Type: application/json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Tue, 08 Mar 2016 22:13:34 GMT
<
* Connection #0 to host localhost left intact
{"discovery":false,"timestamp":1457471908707}
```

Pro detailní zkoumání spojení můžeme využít program Wireshark. Wireshark je svobodný, bezplatný protokolový analyzátor a paketový sniffer.



Obr. 12: Debugování HTTP požadavků ve Wiresharku [autor]

Pro rychlé debugování bez nutnosti dodatečného softwaru lze použít webový prohlížec doplněný o rozšíření schopná posílat HTTP požadavky s různými hlavičkami a metodami.



Obr. 13: Rozšíření "Simple REST Client" pro Google Chrome [autor]

Pro Google Chrome se nám osvědčilo rozšíření Simple REST Client [43]. To je dostupné v oficiálním Internetovém obchodu Chrome. Ostatní prohlížeče mají dostupná podobná rozšíření.

7.2.2. API pro subservery

API pro subservery umožňuje jejich přidávání, mazání a hlavně nejčastěji používané zjišťování informací.

Přehled všech subserverů získáme na:

```
/hausy/api/v1/subservers [HTTP GET, mime=“application/json“]
```

odpovědí je pole:

```
[
 {
 "address":"111",
 "description":"V kuchyni vedle okna u lednice.",
 "location":"U lednice",
 "id":1,
```

```
 "title":"Kuchyn",
 "addDate":1454411993000
 }, ...
]
```

Všechny atributy jsou samovysvětlující. ID je číslo řádku v databázi v tabulce subserverů. Vždy se dotazujeme přes adresu a ne přes id!

Po získání id můžeme získat seznam všech nodů připojených k subserveru:

```
/hausy/api/v1/subservers/111/nodes [HTTP GET, mime="application/json"]
```

Řetězec 111 je id subserveru. Výsledné pole:

```
[
 {
 "wired":false,
 "address":1,
 "channels":3,
 "description":null,
 "location":null,
 "id":184,
 "title":"AUTO_ADD",
 "nodeType":"11",
 "addDate":1457092995000,
 "subserver":"111"
 }
]
```

si podrobněji popíšeme. **wired** je v aktuální verzi nepoužívaný, avšak povinný atribut. V některé z budoucích verzí bude použit k identifikaci bezdrátových nodů. Nyní je vždy na hodnotě **true**.

U nepovinných atributů vrací server **null**. Nejčastěji se s null setkáme u automaticky detekovaných a uživatelem plně nenastavených nodů. Právě nově přidané nody mají implicitně všechy volitelné atributy nastaveny na **null**. **addDate** značí čas přidání nodu ve formátu UNIX Timesamp[9].

9 Počet sekund od půlnoci 1. ledna 1970 UTC.

7.2.3. API pro typy nodů

Při komunikaci mezi všemi vrstvami používáme pro označení typu nodu číslo datového typu bajt. Pomocí tabulky s typy nodů můžeme podle čísla dohledat počet kanálů, zda je nod pro čtení/zápis, analogový/digitální, a další parametry.

Aktuální databázi vždy drží centrální server. Subservery se ho periodicky dotazují na změny a zasílají je C++ aplikaci běžící na Raspberry Pi.

Název	Typ	Delay	Read only	Write only	Data width	Data count
DV_BUTTON	1	false	true	false	1	1
DV_SWITCH	2	false	true	false	1	1
DV_LIGHT	3	false	true	true	1	1
DV_THERMOMETER	6	false	true	false	8	1
DV_PIEZZO	7	false	false	true	1	1
DV_FADER	11	false	true	false	8	1

Tabulka 2: Seznam typů nodů

Seznam získáme na:

```
/hausy/api/v1/nodeTypes [HTTP GET, mime=“application/json“]
```

jako pole:

```
[
 {
 "writeOnly":false,
 "delay":false,
 "description":"Stisk",
 "readOnly":true,
 "id":15,
 "title":"DV_BUTTON",
 "type":"1",
 "dataWidth":1,
 "dataCount":1
 }, ...
]
```

WriteOnly a **readOnly** případně jejich kombinace určují, zda lze z nodu pouze číst, jenom do něho zapisovat nebo kombinaci obojího.

delay značí odložené zpracování. Pokud je nod typu „vstupní tlačítko“, čteme jeho hodnotu okamžitě. Existují však nody, které ke svému změření potřebují větší množství času. Například nod s připojeným teplotním čidlem DS18B20 potřebuje cca 1-2 sekund ke změření hodnot. Pokud bych čekali na odpověď, sběrnice daného subserveru by byla zablokovaná a žádný z ostatních nodů by nemohl komunikovat. Tyto nody pracují se dvěma požadavky. První je požadavek na měření. Nod po jeho přijetí zahájí měření a výsledek si uloží do RAM. Po nějaké době dojde dalším příkazem k vyčtení naměřených hodnot z RAM.

data_width a **data_count** definuje šířku a počet dat. data_width=8 a data_count=3 definuje například jednu RGB LED – 3x8 bitů – 3 bajty.

Nikdy nesmí dojít ke změně stávajícího typu nodu! Pokud by uživatel změnil definici, mohlo by dojít k nefunkčnosti a v krajním případě až poruše systému. Proto se vždy přidávají nové nody. Tím dojde i k zachování zpětné kompatibility s firmware ve starých nodech.

Podle čísla typu nodu zíkáme jeho popis na:

```
/hausy/api/v1/nodeTypes/type/6 [HTTP GET, mime=“application/json“]
```

Číslo 6 je číslem typu nodu, nikoliv jeho ID. ID na jednotlivých subserverech se může lišit, proto ho nikdy nepoužíváme!

Příklad definice typu nodu teploměr:

```
{
 "description":"Teplota",
 "id":20,
 "title":"DV_THERMOMETER",
 "type":"6"
}
```

Pokud bychom potřebovali nod smazat, použijeme stejné URL jako k jeho získání, ale s metodou DELETE:

```
/hausy/api/v1/nodeTypes/type/6 [HTTP DELETE, mime=“application/json“]
```

číslo 6 je opět typ a ne id!

7.2.4. API pro nody

Pro získání JSON pole všech nodů slouží URL:

```
/hausy/api/v1/nodes [HTTP GET, mime="application/json"]
[
 {
 "wired":false,
 "address":1,
 "channels":3,
 "description":null,
 "location":null,
 "id":184,
 "title":"AUTO_ADD",
 "nodeType":"11",
 "addDate":1457092995000,
 "subserver":"111"
 }
]
```

Atributy jsou samovysvětlující. Pro získání konkrétního nodu na základě jeho adresy (ne id) voláme:

```
/hausy/api/v1/nodes [HTTP GET, mime="application/json"]
{
 "wired":false,
 "address":1,
 "channels":3,
 "description":null,
 "location":null,
 "id":184,
 "title":"AUTO_ADD",
 "nodeType":"11",
 "addDate":1457092995000,
 "subserver":"111"
}
```

Seznam všech stavů nodu z databáze získáme na URL:

```
/hausy/api/v1/nodeStates/node/1 [HTTP GET, mime="application/json"]
```

```
{
 "error":"NODE id 1 nema zadne stavy!!!"
}
```

Ke smazání nodu slouží HTTP DELETE metoda na adresu nodu:

```
/hausy/api/v1/nodeStates/node/1 [HTTP DELETE, mime=“application/json“]
```

7.2.5. API pro stavy nodů

Pro získání JSON pole posledních stavů všech nodů slouží URL:

```
/hausy/api/v1/nodeStates [HTTP GET, mime=“application/json“]
[
{
 "address":123,
 "data":[
 {
 "data":[
 0,
 255,
 0
 ],
 "channel":8
 }, ...
 ],
 "message_type":6
},...
]
```

parametry jsou stejné, jako u získání stavu pouze pro jeden nod. Pouze jsou v poli.

Je možné získat pole posledních stavů pouze pro konkrétní subserver a to na URL:

```
/hausy/api/v1/nodeStates/node/ADRESA [HTTP GET, mime=“application/json“]
```

kde adresa je adresou subsystému.

Data se ukládají pro budoucí statistické vyhodnocení, proto není v současné době možné mazání přes API.

7.2.6. API pro pravidla

Pro získání JSON pole všech pravidel slouží URL:

```
/hausy/api/v1/rules [HTTP GET, mime=“application/json“]
[
 {
 "date":1470847797000,
 "evaluationNodeTitle":"AUTO_ADD",
 "evaluationCondition":">=",
 "actionChannel":1,
 "description":"Pri vysoke teplote rozsvit svetlo.",
 "actionNodeAddress":3,
 "title":"Svetlo-teplota",
 "actionNodeTitle":"AUTO_ADD",
 "evaluationNodeAddress":1,
 "actionAddDate":1470847797000,
 "evaluationValue":"27",
 "actionData":"1",
 "id":1,
 "evaluationAddDate":1470847797000,
 "evaluationChannel":1
 } ,...
]
```

je možné se dotázat pouze na jedno pravidlo při znalosti jeho id:

```
/hausy/api/v1/rules/ID [HTTP GET, mime=“application/json“]
```

Id je integer. Výstup je stejný, pouze není v poli.

Na smazání použijeme opět id, ale s metodou delete:

```
/hausy/api/v1/rules/ID [HTTP DELETE, mime=“application/json“]
```

Pro vytvoření je metoda 201 (create):

```
/hausy/api/v1/rules [HTTP CREATE, mime=“application/json“]
```

s daty:

```
{
 "evaluationCondition":">=",
 "actionChannel":1,
```

```
 "description":"Pri vysoke teplote rozsvit svetlo.",
 "actionNodeAddress":3,
 "title":"Svetlo-teplota",
 "actionNodeTitle":"Zarovky",
 "evaluationNodeAddress":1,
 "evaluationValue":"27",
 "actionData":"1",
 "evaluationChannel":1
}
```


7.2.7. API pro podmínky pravidel

Podmínky pravidel slouží k vyhodnocování pravidel mezi daty z jednotlivých nodů. Hausy používá základní podmínky typu větší/menší. Jejich seznam je na:


```
/hausy/api/v1/conditions [HTTP GET, mime="application/json"]
[
 {
 "condition":"<",
 "id":1
 },
 {
 "condition":">",
 "id":2
 },
 {
 "condition":"=",
 "id":3
 },
 {
 "condition":"<=",
 "id":4
 },
 {
 "condition":">=",
 "id":5
 },
 {
```

```
 "condition":"!=",
 "id":6
 }
]
```

Jako vždy je možné získat konkrétní záznam:

```
/hausy/api/v1/conditions [HTTP GET, mime="application/json"]
{
 "condition":"<",
 "id":1
}
```

Protože není důvod podmínky upravovat ani mazat, zpřístupňuje REST API pouze jejich čtení.

7.3. Architektura aplikace

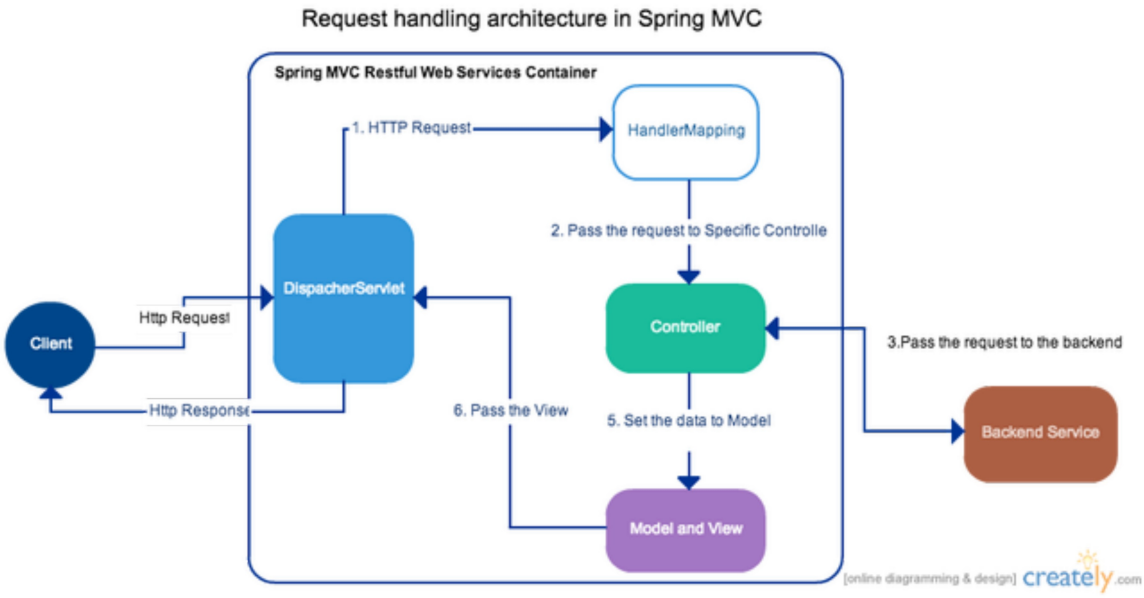
Diplomová práce se zabývá dvěma aplikacemi – centrálního serveru a subserveru. V následujících kapitolách si popíšeme jejich architekturu.

7.3.1. Centrální server

Aplikace centrálního serveru je celá napsána v Javě s využitím frameworků Spring MVC a Hibernate. Pro svůj běh potřebuje webový server a servlet kontejner Apache Tomcat. Využívá upraveného modelu MVC[10]. Požadavky jsou iniciovány pomocí REST API ze subserveru nebo ovládací aplikace. Také mohou přijít požadavky z interního administračního webového rozhraní.

Po přijetí požadavku kontrolerem dojde k jeho vyhodnocení a validaci vstupních parametrů. Poté se v případě potřeby načtou, nebo uloží data z relační databáze. K tomu využíváme objektově relační mapování zprostředkované JPA frameworkem Hibernate. Poté kontroler předá data do view, kde se vygeneruje webová HTML stránka pro administrační rozhraní případně JSON pro REST API.

10 Model-view-controller



Obr. 14: Schéma architektury Spring MVC [44]

Centrální server využívá Spring IoC[11] pro dependency injection. Díky IoC a JPA je možné vyměnit části aplikace za jiné s minimální nutností úpravy kódu. Velká výhoda dependency injection je také při psaní testů.

7.3.2. Lokální server (subserver)

Lokální server ja také celý psaný v Javě, ale na rozdíl od centrálního serveru neběží v servletovém kontejneru, ale jako samostatná aplikace.

Protože v případě výpadku systému přebírá funkci centrálního serveru, bylo použito maximální množství kódu právě z centrálního serveru. Využíváme stejné frameworky a to Spring MVC, Spring IoC a Hibernate.

Po startu se spustí dvě třídy *Communication* a *REST*. Každá v samostatném vlákně a běží po celou dobu běhu aplikace. Obě třídy jsou singleton.

Třída *Communication* se ihned po svém vytvoření připojí přes TCP socket k C++ aplikace a pomocí ní komunikuje s koncovými nody. Je schopná získávat data z nodů a naopak jim data posílat. Data si ukládá ve své MySQL databázi. Pokud potřebuje poslat data na centrální server, využívá instanci třídy *REST*.

11 Inversion of Control

Třída *REST* slouží pro komunikaci s centrálním serverem. Po vytvoření instance se nejdřív spojí s centrálním serverem. Získá si data o dostupných nodech a jejich pravidlech. Pokud se liší, aktualizuje si vlastní databázi. Při vyhledávání nových nodů přepošle požadavek z centrálního serveru přes třídu Communication a poté během párování přeposílá komunikace mezi centrálním serverem a C++ aplikací, respektive nody samotnými.

Jakmile dojde k vyčtení dat z nodů, ihned se pošlou na server a uložení do lokální databáze. Na lokálně uložená data je možné v případě výpadku centrálního serveru aplikovat pravidla pro řízení nodů.

Další důležitou třídou je třída *RuleEvaluate*. V té probíhá samotné vyhodnocování pravidel. Třída si načte poslední naměřená data z databáze a aplikuje na ně pravidla. Po jejich vyhodnocení pošle přes třídu *Communication* řídící data nodům.

8. Závěr

Cílem práce bylo vytvořit ovládací software pro řízení chytrého domu. Byly vytvořeny dvě aplikace v programovacím jazyku Java. Jedna pro centrální server a druhá pro servery lokální. Centrální server řídí data ze všech lokálních serverů a pomocí REST API přijímá povely od uživatele z mobilních a webových aplikací. Lokální server komunikuje přímo s C++ vrstvou řídící koncové nody. Při výpadku centrálního serveru nebo spojení s ním převezme lokální server dočasně jeho roli. Funkcionalita systému je sice omezená, ale uživatel má funkční dům do doby opravení závady.

Po srovnání existujících řešení v 2. kapitole jsme zjistili, že neexistuje systém, který by plně vyhovoval našim požadavkům, a proto jsme se rozhodli navrhnout a implementovat vlastní. Ve 3. kapitole jsme si popsali požadavky a popsali architekturu systému. Podle návrhu jsme ve 4. kapitole vybrali programovací jazyk, frmeworky a databázi. Na závěr jsme porovnali existující jednodeskové počítače a vybrali Raspberry Pi. V 5. a 6. kapitole jsme podrobně popsali poskytovanou funkcionalitu a fungování centrálního serveru a subserveru. V poslední 7. kapitole je podrobně popsané REST API včetně popisu zasílaných zpráv všech vytvořených aplikací. Jedná se o API pro komunikaci mezi centrálním serverem a subserverem, subserverem a aplikací řídící hardwarové moduly (nody) a také API pro mobilní aplikace.

Během testování se ukázala velká hardwarová náročnost Javy v kombinaci se Spring frameworkem. Systém funguje, ale pokud by bylo připojeno velké množství subserverů a nodů, například ve velké kancelářské budově, nastane značné zpomalení. Řešením by bylo použít výkonnější počítače na řízení, nebo vybrat méně náročný jazyk a aplikaci do něj přepsat. Díky využívání API je možné jednotlivé části systému (vrstvy) libovolně zaměňovat bez omezení funkčnosti.

Systém je možný provozovat v klasickém schématu – jeden centrální server a více lokálních serverů s připojenými nody. Pokud by bylo potřeba, například v malém bytě, je možné provozovat všechny vrstvy aplikace na jednom počítači a ušetřit tak náklady na pořízení hardwaru.

Do budoucna by bylo dobré použít místo RESTu socketové spojení mezi centrálním serverem a lokálními servery. Zjednodušila by se komunikace a zvýšila její rychlost. Vhodnou technologií by mohly být třeba WebSockety. Také by bylo dobré zdokonalit validaci dat a oprávnění. V aktuální verzi je spojení mezi všemi vrstvami nešifrované. Kvůli větší bezpečnosti by bylo vhodné zavést SSL/TLS šifrování mezi jednotlivými aplikacemi.

9. Seznam obrázků

10. Seznam tabulek

11. Použité zdroje

1. ŠTĚPÁN, Jan. Návrh a implementace HW a SW smart zařízení pro komunikaci v inteligentním domě. Hradec Králové, 2015. Diplomová práce. Univerzita Hradec Králové. Vedoucí práce Mgr. Josef Horálek, Ph.D.
2. Domácí automatizace I [online]. [cit. 2016-08-21]. Dostupné z: http://vyvoj.hw.cz/teorie-a-praxe/konstrukce/domaci-automatizace-i.html
3. SMIREK, Lukas; ZIMMERMANN, Gottfried; ZIEGLER, Daniel. Towards universally usable smart homes-how can myui, urc and openhab contribute to an adaptive user interface platform. IARIA, Nice, France, 2014, 29-38.
4. LEONG, Chui Yew; RAMLI, Abdul Rahman; PERUMAL, Thinagaran. A rule-based framework for heterogeneous subsystems management in smart home environment. IEEE Transactions on Consumer Electronics, 2009, 55.3: 1208-1213.
5. VALTCHEV, Dimitar; FRANKOV, Ivailo. Service gateway architecture for a smart home. IEEE Communications Magazine, 2002, 40.4: 126-132.
6. KUMAR, Shiu. Ubiquitous smart home system using android application. arXiv preprint arXiv:1402.2114, 2014.
7. KAILA, Lasse; VAINIO, Antti-Matti; VANHALA, Jukka. Connecting the smart home. In: Proceedings of IASTED International Conference on Networks and Communication Systems (NCS 2005), Krabi, Thailand. 2005.
8. CHONG, Gao; ZHIHAO, Ling; YIFENG, Yuan. The research and implement of smart home system based on internet of things. In: Electronics, Communications and Control (ICECC), 2011 International Conference on. IEEE, 2011. p. 2944-2947.
9. Open Source Initiative [online]. [cit. 2016-08-21]. Dostupné z: https://opensource.org/
10. OpenHAB a jeho instalace na Raspberry Pi. Root.cz [online]. 2015 [cit. 2016-08-21]. ISSN 1212-8309. Dostupné z: http://www.root.cz/clanky/openhab-a-jeho-instalace-na-raspberry-pi/
11. Java.com: Java + You [online]. [cit. 2016-08-21]. Dostupné z: https://www.java.com/en/

12. Raspberry Pi - Teach, Learn, and Make with Raspberry Pi [online]. [cit. 2016-08-21]. Dostupné z: https://www.raspberrypi.org/
13. Scratch - Imagine, Program, Share [online]. [cit. 2016-08-21]. Dostupné z: https://scratch.mit.edu/
14. MENS, Jan-Piet. A story of home automation with openHAB, Z-Wave, and MQTT [online]. [cit. 2016-08-21]. Dostupné z: http://jpmens.net/2014/01/14/a-story-of-home-automation/
15. Z-Wave Home control | Z-Wave Smart Home [online]. [cit. 2016-08-21]. Dostupné z: http://www.z-wave.com/
16. Domoticz [Project: Turris | Gadgets] [online]. [cit. 2016-08-21]. Dostupné z: https://www.turris.cz/gadgets/domoticz
17. Domoticz [online]. [cit. 2016-08-21]. Dostupné z: https://domoticz.com/
18. Produkty ELKO EP, s.r.o. [online]. [cit. 2016-08-21]. Dostupné z: http://www.elkoep.cz/produkty/inels-bus-system/novinky/
19. INELS: Váš dům Vás má rád [online]. [cit. 2016-08-21]. Dostupné z: http://www.elkoep.cz/downloads/promotion_materials/iNELS_pro_laiky_30.pdf
20. Inteligentní ovládání hudby a audia v celém domě - Loxone [online]. [cit. 2016-08-21]. Dostupné z: http://www.loxone.com/cscz/chytry-dum/moznosti-vyuziti/hudba.html
21. Jednoduchá a levná inteligentní elektroinstalace - Loxone [online]. [cit. 2016-08-21]. Dostupné z: http://www.loxone.com/cscz/start.html
22. Vendor lock-in definition by The Linux Information Project (LINFO) [online]. [cit. 2016-08-21]. Dostupné z: http://www.linfo.org/vendor_lockin.html
23. ATR [online]. [cit. 2016-08-21]. Dostupné z: http://www.e-automatizace.cz/ebooks/ridici_systemy_akcni_cleny/Akc_cleny.html
24. Bluetooth Low Energy | Bluetooth Technology Website [online]. [cit. 2016-08-21]. Dostupné z: https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy

25. TIŠNOVSKÝ, Pavel. Sběrnice RS-422, RS-423 a RS-485 - Root.cz [online]. [cit. 2016-08-21]. Dostupné z: http://www.root.cz/clanky/sbernice-rs-422-rs-423-a-rs-485/

26. Cross platform, open source .NET framework [online]. [cit. 2016-08-21]. Dostupné z: http://www.mono-project.com/

27. Price Lists | Global Pricing and Licensing [online]. [cit. 2016-08-21]. Dostupné z: http://www.oracle.com/us/corporate/pricing/price-lists/index.html

28. Vaadin: QuickTickets Dashboard [online]. [cit. 2016-08-21]. Dostupné z: http://demo.vaadin.com/dashboard/

29. MALÝ, Martin. REST: architektura pro webové API - Zdroják [online]. [cit. 2016-08-21]. Dostupné z: https://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/

30. HANEL, David. Kapitola 3. Úvod do JavaServer Faces [online]. [cit. 2016-08-21]. Dostupné z: http://java.vse.cz/jsf/chunks/ch03.html

31. HAUER, Philipp. Evaluating Vaadin: Strengths and Weaknesses [online]. 2015 [cit. 2016-08-21]. Dostupné z: http://blog.philipphauer.de/evaluating-vaadin-strengths-weaknesses/

32. MinnowBoard Wiki [online]. [cit. 2016-08-21]. Dostupné z: http://wiki.minnowboard.org/MinnowBoard_Wiki_Home

33. Raspberry Pi 2 on sale now at $35 [online]. [cit. 2016-08-21]. Dostupné z: https://www.raspberrypi.org/blog/raspberry-pi-2-on-sale/

34. STEHLÍK, Petr. Linux na Orange Pi Plus: SATA je chyták - Root.cz [online]. 2016 [cit. 2016-08-21]. Dostupné z: http://www.root.cz/clanky/linux-na-orange-pi-plus-sata-je-chytak/

35. FreeBSD/ARM on Raspberry Pi [online]. [cit. 2016-08-21]. Dostupné z: https://wiki.freebsd.org/FreeBSD/arm/Raspberry%20Pi

36. KOSEK, Jiří. Využití webových služeb a protokolu SOAP při komunikaci [online]. [cit. 2016-08-21]. Dostupné z: http://www.kosek.cz/diplomka/html/websluzby.html

37. Representational State Transfer [online]. [cit. 2016-08-21]. Dostupné z: https://goo.gl/IZ7AW9

38. ANDREI, Alin. Oracle Java 8 (Stable) Released, Install it In Ubuntu ~ Web Upd8: Ubuntu / Linux blog [online]. [cit. 2016-08-21]. Dostupné z: http://www.webupd8.org/2014/03/oracle-java-8-stable-released-install.html
39. NEAL, Marcas. How do I deploy a WAR file in tomcat 7 in Ubuntu? [online]. [cit. 2016-08-21]. Dostupné z: https://www.quora.com/How-do-I-deploy-a-WAR-file-in-tomcat-7-in-Ubuntu
40. Úvod do JSON [online]. [cit. 2016-08-21]. Dostupné z: http://www.json.org/json-cz.html
41. RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format [online]. 2014 [cit. 2016-08-21]. Dostupné z: https://tools.ietf.org/html/rfc7159
42. WALSH, David. POST Form Data with cURL [online]. 2016 [cit. 2016-08-21]. Dostupné z: https://davidwalsh.name/curl-post-file
43. Simple REST Client - Internetový obchod Chrome [online]. [cit. 2016-08-21]. Dostupné z: https://chrome.google.com/webstore/detail/simple-rest-client/fhjcajmcbmldlhcimfajhfbgofnpcjmb?utm_source=chrome-app-launcher-info-dialog
44. MALALANAYAKE, Dinuka. Spring MVC with Hibernate. Complex to Simple [online]. [cit. 2016-08-21]. Dostupné z: https://malalanayake.wordpress.com/2014/07/27/spring-mvc-with-spring-hibernate/
45. OpenHAB Screenshots [online]. [cit. 2016-08-21]. Dostupné z: https://www.mysensors.org/controller/openhab

Přílohy

Součástí diplomové práce je přiložené CD se zdrojovými kódy aplikace centrálního serveru a lokálního serveru. Na CD se nacházejí adresáře se zdrojovými kódy a binární soubory obou aplikací připravené k nasazení. V případě centrálního serveru jde o *war* soubor pro nasazení v Apache Tomcat. Aplikace lokálního serveru se sestává z *jar* soubor.

FIM UHK
UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Rokitanského 62, 500 03 Hradec Králové, tel: 493 331 111, fax: 493 332 235

Zadání k závěrečné práci

Jméno a příjmení studenta: **Martin Vancl**

Obor studia: Aplikovaná informatika (2)
Jméno a příjmení vedoucího práce: **Josef Horálek**

Název práce:
Platformy pro domácí automatizaci s využitím inteligentních prvků

Název práce v AJ:
Platforms for home automation with intelligent features

Podtitul práce:

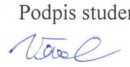
Podtitul práce v AJ:

Cíl práce: Cílem práce je navrhnout a realizovat část zařízení pro komunikaci v inteligentním domě na úrovni logické správy systému. Komunikace a vyhodnocování dat z příslušných subsystémů.

Osnova práce:
Úvod
Rešerše dostupných přístupů
Představení komplexního modelu systému
Logický návrh komunikace se subsystémy
Logický návrh úloh serveru
Výběr vhodných SW součástí serveru
Představení realizovaného řešení
Závěr

Projednáno dne:

Podpis studenta



Podpis vedoucího práce