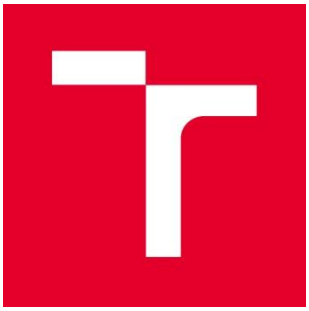


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

INTERNET VĚCÍ S UZLY NA BÁZI PNVM

INTERNET OF THINGS WITH PNVM-BASED NODES

DIPLOMOVÁ PRÁCE
MATER'S THESIS

AUTOR PRÁCE — Bc. TOMÁŠ KOREJTKO
AUTHOR

VEDOUCÍ PRÁCE — Doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Korejtko Tomáš, Bc.**
Obor: Inteligentní systémy
Téma: **Internet věcí s uzly na bázi PNVM**
Internet of Things with PNVM-Based Nodes
Kategorie: Softwarové inženýrství

Pokyny:

1. Seznamte se s problematikou Internet of Things (IoT) a prostudujte související otevřené technologie. Zaměřte se přitom na aplikace v oblasti Smart Home, na platformu Raspberry Pi a na protokol MQTT.
2. Seznamte se s aktuální implementací PNVM (Petri Net Virtual Machine) a s konceptem potenciálních aplikací PNVM.
3. Navrhněte způsob využití využití PNVM v IoT za pomoci existujících i nově navržených prostředků kompatibilních s MQTT (databáze, dashboard pro různé typy uživatelů, prostředky pro tvorbu, údržbu a rekonfiguraci systému).
4. Navržený systém realizujte a ověřte jeho funkčnost na vhodné demonstrační aplikaci z oblasti Smart Home.

Literatura:

- RICHTA Tomáš a JANOUŠEK Vladimír. Operating System for Petri Nets-Specified Reconfigurable Embedded Systems. In: Computer Aided Systems Theory - EUROCAST 2013. Berlin Heidelberg: Springer Verlag, 2013, s. 444-451. ISBN 978-3-642-53855-1.
- RENEW. http://www.renew.de/
- MQTT. http://mqtt.org/

Při obhajobě semestrální části projektu je požadováno:

- Body 1, 2 a část návrhu.

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

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

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

Vedoucí: **Janoušek Vladimír, doc. Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

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

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Práce se zabývá problematikou Internet of Things (IoT) a otevřených technologických řešení s tím souvisejících. Konkrétně se zaměřuje na softwarová řešení v oblasti chytrého domu kompatibilní s platformou Raspberry Pi a protokolem MQTT. Dále se práce zabývá studiem Petri Net Virtual Machine (PNVM) a jeho potenciální aplikace v IoT. Cílem práce je navrhnout integraci PNVM v IoT za pomoci existujících prostředků kompatibilních s MQTT a implementovat demonstrační aplikaci pro chytrý dům.

Klíčová slova

Internet věcí, Raspberry Pi, MQTT, Petriho sítě, chytrý dům, softwarové inženýrství

Abstract

This thesis focuses on Internet of Things (IoT) and open-source technologies based on it. Specifically aims at software solutions relevant to smart home and compatible with Raspberry Pi platform and MQTT communication protocol. This thesis also focuses on studying Petri Net Virtual Machine (PNVM) and its potential application in IoT. The objective is to design integration of PNVM into IoT with help of existing software means compatible with MQTT and implement a demo application for smart home.

Keywords

Internet of Things, Raspberry Pi, MQTT, Petri nets, smart home, software engineering

Citace

KOREJTKO Tomáš. *Internet věcí s uzly na bázi PNVM*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. Vladimír Janoušek, Ph.D..

Internet věcí s uzly na bázi PNVM

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Doc. Ing. Vladimíra Janouška, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Bc. Tomáš Korejtko
24.5. 2017

Poděkování

Děkuji docentu Vladimíru Janouškovi za poskytnutí zajímavého tématu, programů potřebných pro vznik této práce a za cenné rady a připomínky, které mě vedly správným směrem.

Obsah

1 Úvod

Technologie se stále rozrůstají, stále přibývá elektronických a chytrých zařízení. Dnes už téměř každý vlastní chytrý telefon a každý je připojený k internetu. A to neplatí jen pro domácnosti, existují například chytré křižovatky, které ovlivňují plynulost provozu. Vše funguje na jednoduchém principu, kdy senzory či snímače čtou aktuální data z prostředí, která se vyhodnotí v řídícím kontroléru a ten ovládá aktuátory, které nějakým způsobem ovlivňují prostředí, kde se nacházejí, nebo ovládají nějaký přístroj.

Důležitou součástí této práce je PNVM (Petri Net Virtual Machine), což je součást operačního systému pro konkrétní mikrokontrolér nebo PC, a jeho úkolem je interpretovat model složený z Petriho sítí. Bude nutné prostudovat aktuální implementaci PNVM, seznámit se s referenčními Petriho sítěmi a s tím spojené možnosti dynamické rekonfigurace, což je jedním z hlavních cílů PNVM (více o PNVM v kapitole 3).

V této práci se zaměříme na internet věcí (IoT) a především na oblast chytrých domů. Především na platformu Raspberry Pi (více o Raspberry v kapitole 2.2) a na protokol MQTT (více o MQTT v kapitole 2.3). V této části také projdeme několik kompletních řešení pro chytrý dům.. Poté důkladně prozkoumáme PNVM, jeho koncepty a principy, a také konkrétní implementační detaily umožňující jeho integraci s Raspberry Pi a protokolem MQTT. Následně prozkoumáme existující otevřený software, díky kterému sestavíme systém obsahující Raspberry Pi, PNVM a protokol MQTT. Tento software zahrnuje databáze, různé dashboardy a softwarová řešení s tím související, která ve výsledku umožní přehlednou správu chytrého domu. Nakonec vybereme prostředky vhodné pro integraci PNVM do IoT a navrhneme způsob jejich využití a na základě návrhu implementujeme demonstrační aplikaci pro chytrý dům.

2 Internet věcí

Jak jsem již zmínil v úvodu, IoT se týká propojení a komunikace elektronických zařízení mezi sebou a to typicky bez nutnosti častého vstupu od člověka. V této kapitole se zaměříme rovnou na věci relevantní našim cílům. V kapitole 2.1 se budeme věnovat chytrému domu, možnosti automatizace v domě a několika konkrétním softwarovým řešením. Následující kapitola 2.2 již přejde ke konkrétnímu hardwaru, na kterém poběží náš software a zároveň sem bude možno připojit senzory či aktuátory. V poslední kapitole 2.3 popíšeme komunikační protokol MQTT, jeho vlastnosti a důvod použití právě tohoto protokolu.

2.1 Chytrý dům

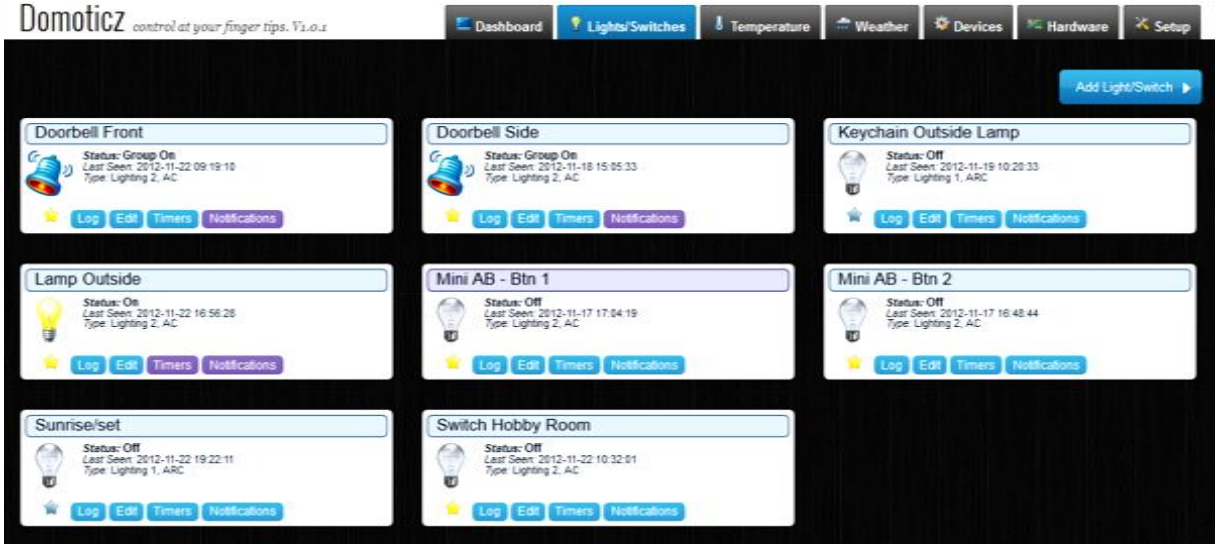
Možností automatizace v domě je nepřeberné množství - například rolování žaluzií podle denní doby nebo podle míry přicházejícího světla, regulace teploty v pokoji otevíráním a zavíráním topení nebo dokonce simulace přítomnosti člověka v době dovolené.

Chytrý dům tedy musí mít hardwarovou část (senzory, aktuátory) a softwarovou část (řízení). Věnujme se pouze softwarové řídící části. Ta musí vědět o senzorech i aktuátorech, to znamená, že si musí pamatovat nějaký seznam adres reprezentující jednotlivé prvky. A samozřejmě musí obsahovat samotnou logiku řízení. Řídící software také typicky umožňuje uživateli konfigurovat hardwarové i jiné prvky a definovat řídící logiku přes webové rozhraní.

Některá existující open-sourcová řešení domácí automatizace:

- Domoticz
- OpenHAB

Domoticz je open-sourcový systém domácí automatizace implementovaný v jazyce C/C++. Umožňuje přidávat senzory a aktuátory s podporou různých komunikačních protokolů. Při troše nastavení dokáže podporovat i MQTT. Programování logiky řízení je řešeno skládáním blokového schématu, kde lze definovat podmínky a akce. To vše přes přehledné webové rozhraní. Domoticz velice pěkně umí zobrazit historii senzorů ve formě grafu, problém je v tom, že starší data se komprimují, tudíž při pohledu dále do historie jsou data méně přesná. Někdy to může být i výhoda, zvláště při použití například na Raspberry Pi, kde je úložiště omezené. Bohužel tato vlastnost nejde přenastavit. Další nevýhodou je, že každý senzor nebo aktuátor se v Domoticz používá pomocí id, které je vygenerované při přidání senzoru nebo aktuátoru přes webové rozhraní, což ztěžuje aplikaci dynamické rekonfigurace. [1]



Obrázek 2.1: Ukázka Domoticz webového rozhraní[1]

OpenHAB (Open Home Automation Bus) je open-sourcový nástroj pro integraci řešení domácí automatizace napsaný v jazyce Java na platformě OSGi.

OSGi platforma definuje dynamický komponentní model pro Javu, jeho jednotkou je bundle, což je název pro modul. Důležité je, že OSGi platforma podporuje nahrávání bundlů za chodu systému bez nutnosti restartu, což je velmi přínosné pro dynamickou rekonfiguraci.

OpenHAB runtime je řízen třemi hlavními konfiguračními soubory:

- *items* - definuje takzvaný itemy neboli položky, základní stavební prvky OpenHABu, které representují konkrétní prvky chytrého domu. Může se jednat například o číslo nebo string pro senzory a nebo přepínač či posuvník pro aktuátory.
- *rules* - pravidla, která se skládají z podmínky a akce. Podmínka může být složená a může detekovat změnu nějakého itemu nebo i například start samotného OpenHABu a na to reagovat nějakou akcí nebo více akcemi.
- *sitemaps* - zde je definováno rozložení webového rozhraní. Konkrétně jde tedy o rozložení itemů a způsob jejich zobrazení.

Tyto konfigurační soubory jsou klasické plain textové soubory, které je třeba buď ručně modifikovat nebo použít OpenHAB designér, který je specificky vytvořený pro tvorbu konfigurace OpenHABu včetně zvýrazňování syntaxe a kontextové nápovědy. Konfigurace přes webové rozhraní není možná. Tento druh konfigurace už ale zastiňuje potenciál OSGi, protože je vždy nutné ručně editovat konfigurační soubory. Výhodou OpenHABu z hlediska OSGi je možnost lehce přidat hardware podporující nový komunikační protokol, protože pokud pro něho existuje binding (bundle starající se o komunikaci s hardwarem), stačí ho pouze přidat do OpenHAB runtimu i za běhu systému a můžeme ho hned používat. Mezi bindingy, které jsou standardní součástí OpenHABu patří i MQTT binding, takže podpora MQTT není problém. [2]

[1] https://a.fsdn.com/con/app/proj/domoticz/screenshots/Domoticz.png/1

Obrázek 2.2: Ukázka OpenHAB webového rozhraní[1]

2.2 Raspberry Pi

Raspberry Pi je mini-počítač původně vyvinut za účelem výuky. Jeho nízké náklady a relativně vysoký výkon z Raspberry Pi dělají ideální prostředek pro chytrý dům. Má operační systém na bázi linuxu, typicky se jedná o Raspbian, ale lze nainstalovat jakýkoliv systém pro ARM procesory. Raspberry se vyrábí od roku 2012 a za tu dobu se dočkal velkého zvýšení výkonu a dalších vylepšení, což je patrné z následující tabulky.

Tabulka 2.1: Přehled modelů Raspberry Pi[2]

Model	Pi 1 A+	Pi 2 B	Pi 3
CPU	ARM1176JZF-S (32 bit) 1-jádro @ 700 MHz	ARM Cortex-A7 (32 bit) 4-jádro @ 900 MHz	ARM Cortex-A53 (64 bit) 4-jádro @ 1200 MHz
RAM	256 MB	1 GB	1 GB
USB portů	1	4	4
Síť	žádná	10/100 Mbit/s Ethernet USB adaptér pro wifi	10/100 Mbit/s Ethernet 802.11n wi-fi Bluetooth 4.1
Napájení	5 V micro-usb 300 mA (1.5 W)	5 V micro-usb 600 mA (3.0 W)	5 V micro-usb 800 mA (4.0 W)

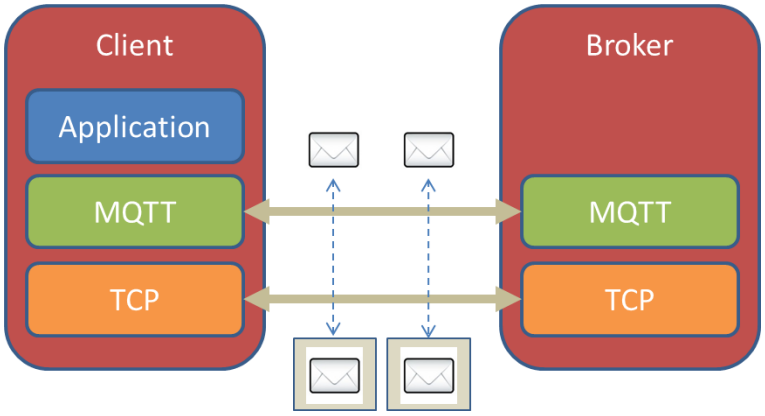
[1] http://www.openhab.org/assets/images/ui/classicui.png
[2] https://en.wikipedia.org/wiki/Raspberry_Pi#Specifications

2.3 Protokol MQTT

Jinak také Message Queue Telemetry Transport je komunikační protokol na bázi publish/subscribe. MQTT pracuje nad protokolem TCP/IP. Je navržen tak, aby měl co nejmenší velikost paketů, a je vhodný především pro místa, kde je vysoká odezva sítě, malá propustnost nebo obecně pro nespolehlivé sítě. MQTT také obsahuje QoS (quality of service) pro doručování zpráv, a to:

- "Nanejvýš jednou" - zde je připuštěno, že zpráva se ztratí a nebude doručena. To se hodí například v situaci, kdy nějaký senzor čte data třeba několikrát za sekundu, takže jedna zpráva nebude chybět, protože následuje za krátko další.
- "Nejméně jednou" - tady je nutné, aby zpráva dorazila jednou, a může dojít i k duplikaci.
- "Přesně jednou" - zpráva musí dojít přesně jednou. Dá se použít například v bankovnictví, kde duplicita zprávy by mohla vést ke špatným výsledkům.

MQTT disponuje vlastností "Last Will", což znamená, že při ztrátě spojení s klientem broker publikuje o této situaci zprávu, a kdo se přihlásil k jejímu odběru, ten ji dostane. Z toho všeho plyne, že MQTT je velmi vhodný pro použití v IoT, a za tímto účelem byl také navrhován. [3]



Obrázek 2.3: Architektura MQTT[1]

Odběr a publikování v MQTT je adresováno přes *topicy*, které mají podobný tvar jako cesty v souborovém systému. Jde tedy o slova oddělená lomítkem. Klient se hlásí k odběru konkrétního *topicu* nebo více *topiců* a dostává zprávy publikované právě na tyto *topicy*. MQTT také umožňuje při odebírání *topicu* použít *wildcards*, tedy zástupné znaky. Tím pádem lze přihlášením k jednomu *topicu* se zástupnými znaky odebírat více *topiců* najednou. Zástupné znaky jsou: [4]

- + -zástupný znak pro jednu položku *topicu*, kde položkou se rozumí část mezi dvěma lomítky nebo první část před prvním lomítkem nebo poslední část za posledním lomítkem.
- # - zástupný znak pro více položek *topicu* po sobě následujích.

Mějme například odběr skládající se z následujících *topiců* se zástupným znakem +, které přijmou *topic* "a/b/c/d":

- a/b/c/d

[1] http://www.embedded101.com/Portals/0/images/DNNArticle/Windows-Live-Writer/378e0d5a6034_DE32/Fig3.2_4.png

- +/b/c/d
- a/+/c/d
- a/+/+/d
- +/+/+/+

Následující odběr *topiců* naopak nepřijme "a/b/c/d":

- a/b/c
- b/+/c/d
- +/+/+

Pokud vezmeme v úvahu odběr *topiců* se zástupným znakem #, budou následující odběry přijímat *topic* "a/b/c/d":

- a/b/c/d
- #
- a/#
- a/b/#
- a/b/c/#
- +/b/c/#

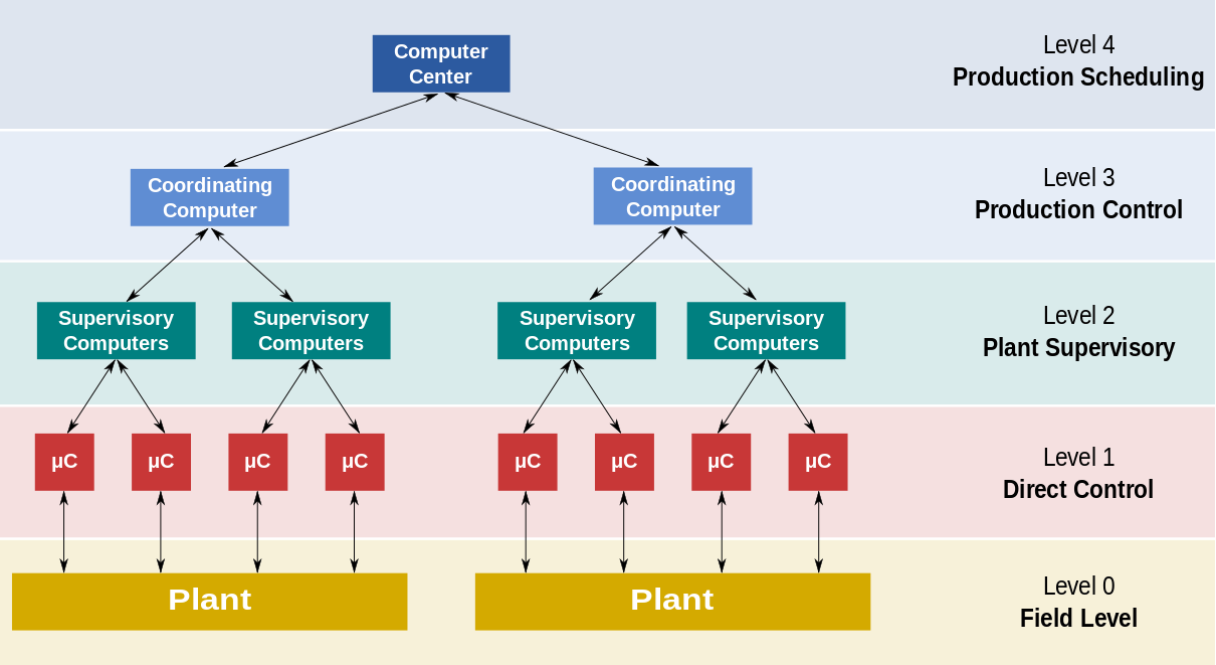
MQTT má také speciální *topicy*, na kterých odesílá systémové informace týkající se MQTT brokeru. Tyto *topicy* mají jako první položku $SYS. Například pokud se klient přihlásí k odběru *$SYS/broker/clients/total*, bude pravidelně dostávat informaci o celkovém počtu klientů připojených k danému brokeru. Nebo odběrem *$SYS/broker/messages/sent* by dostávali informaci o celkovém počtu odeslaných zpráv od doby, kdy se broker spustil.

MQTT standard nepředepisuje normu pro bezpečnost, ale silně doporučuje mít prostředky pro autentizaci/autorizaci. A také alespoň ty nejznámější brokery jako Mosquitto nebo Mosca ji implementují. Defaultní je pro broker nemít žádnou autentizaci, ale lze ji v konfiguraci zapnout a je to doporučeno. Nejjednodušší autentizační metodou je použití *ACL* souboru, tedy souboru určujícího ovládání přístupu uživatelů podle jména a hesla. V tomto souboru může být nastaveno, který uživatel bude moct odebírat který *topic* a stejně tak, na který *topic* může publikovat. V takovémto případě musí klient připojující se k brokeru zadat platné jméno a heslo, aby měl přístup k *topicům* definovaným v *ACL* souboru. Komunikaci MQTT brokeru jde také zabezpečit pomocí SSL/TLS certifikátů. Tím pádem už klient při přihlašování nemusí zadávat jméno a heslo, ale musí mít platný certifikát.

Další vlastností MQTT brokerů je možnost vzájemného propojení. To znamená, že je možné nastavit, aby více MQTT brokerů bylo mezi sebou hierarchicky propojeno. Vždy jeden broker definuje "most" na druhý broker, kde vybere které *topicy* se budou přeposílat na další broker podle prefixu *topicu* a případně na jaký prefix se budou tyto *topicy* mapovat na vzdáleném brokeru. Toto spojení funguje oběma směry. Pokud vzdálený broker publikuje zprávu s prefixem, který definuje první broker jako prefix pro vzdálený broker, dostane ho lokální broker a opět bude přemapovaný na lokální prefix. [3][4]

2.4 SCADA systémy

SCADA systémy se týkají spíše průmyslu než chytrého domu, ale principy se používají podobné. SCADA je architektura ovládacího systému skládajícího se ze softwaru i hardwaru, která má za úkol ovládat lokální nebo i vzdálené procesy, monitorovat, sbírat a zpracovávat *real-time* data, přímo komunikovat se zařízením jako je například senzor nebo aktuátor skrz HMI (*human-machine interface*), ale i tvořit logy z událostí. [5]



Obrázek 2.4: Úrovně SCADA systému[1]

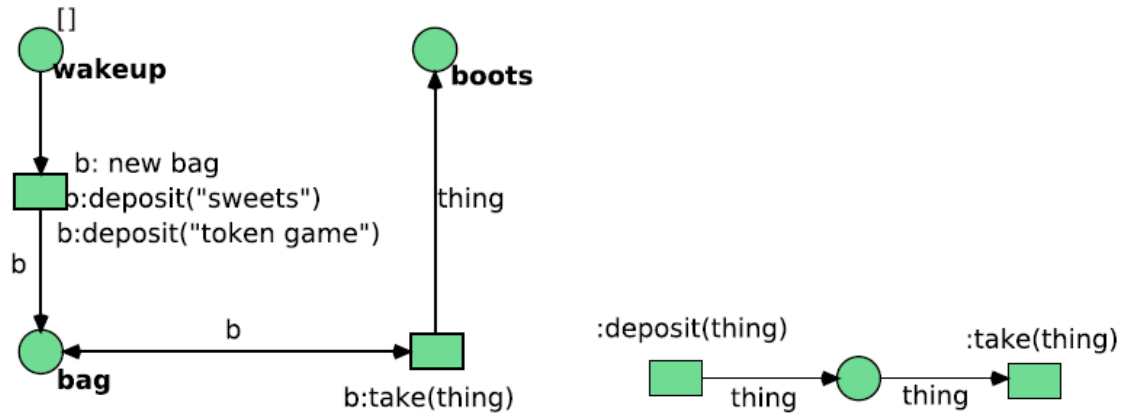
- Úroveň 0 - zde se nacházejí senzory a aktuátory.
- Úroveň 1 - obsahuje microkontroléry pro sběr dat. U SCADA systému jde typicky o PLC nebo RTU.
- Úroveň 2 -místo pro SCADA systém a výpočetní platformu. Zde se také nacházejí například setpointy, které poté použije úroveň 1.Na úrovni 1 se nachází algoritmy pro řízení a jejich parametry řeší úroveň 2.
- Úroveň 3 - je úroveň, která už neovládá procesy, ale spíše zahrnuje monitorování a business logiku.
- Úroveň 4 - je úroveň pro plánování.

[1] By Daniele Pugliesi - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=31527335

3 Petri Net Virtual Machine

3.1 Referenční Petriho sítě

Základním konceptem PNVM jsou Referenční Petriho sítě. Ty jsou založeny na myšlence "nets-within-nets" neboli sítě v sítích. To umožňuje sítím být zanořené v jiných sítích ve formě tokenů. V sítích jsou vzájemně propojená místa a přechody. Místa mohou obsahovat tokeny. Pokud je v místě před přechodem dostatek tokenů, přechod se může provést, což atomicky změní stav sítě. Tokeny jsou odebrány z míst před přechodem a vloženy do míst za přechodem. Přechod také může mít "guard", tj. strážní podmínku, která omezuje možnost provedení přechodu. Referenční sítě jsou schopné komunikovat s vnořenými sítěmi pomocí *downlinků* a *uplinků*. Uplink je speciální druh přechodu, který může být proveden pouze jako součást jiného přechodu nadřazené sítě, který ho zavolá. Downlink jde naopak z nadřazené sítě do vnitřní. Znázornění na následujícím obrázku. [6]

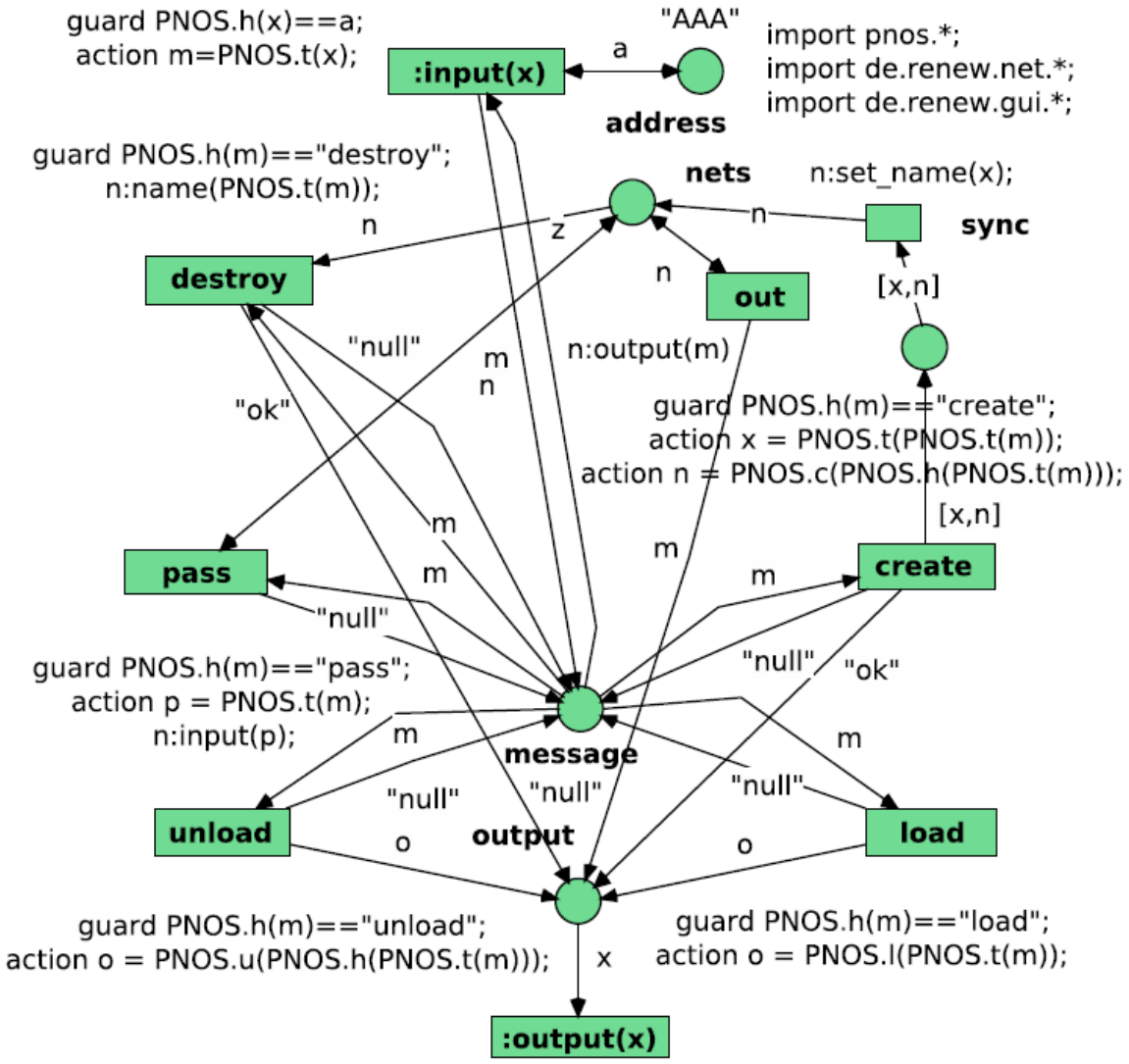


Obrázek 3.1: Přiklad referenční sítě [7]

Na obrázku vidíme 2 referenční sítě. Síť (a) ve svém prvním přechodu vytvoří novou instanci sítě (b) a pomocí downlink *b:deposit* vloží 2 věci do batohu. V síti (b) vidíme, že obsahuje 2 uplinky, a to *:deposit* a *:take*. V druhém přechodu sítě (a) se berou věci ze sítě (b) a vkládají se do místa *boots*.

3.2 Současná implementace PNVM

Aktuálně je PNVM implementované pro PC s operačním systémem linux a architekturou x86. V úvodu jsem zmínil, že PNVM běží jako aplikace operačního systému. Těchto PNVM aplikací může běžet více současně a mohou mezi sebou komunikovat. Komunikace je momentálně řešena pomocí linuxových *pipes*. Samotné PNVM je řízené referenční sítí, díky které je možné nahrávat a instanciovat další sítě. Hlavní síť PNVM je zobrazena na následujícím obrázku.



Obrázek 3.2: Původní návrh hlavní sítě PNVM [6]

Z obrázku lze vyčíst, že síť má uplink pro vstup PNVM, který přichází přes linuxovou *pipu* a uplink pro výstup, který vede na standardní výstup. Také můžeme vidět, jaké příkazy PNVM podporuje a jak fungují, obecně jsou ve tvaru <*adresa*> <*příkaz*> <*data*>, kde adresa je název sítě pro který je příkaz určený, příkaz je určení přechodu sítě který se má provést, a data obsahují parametry volání. V případě příkazu *pass* jsou tyto parametry opět ve tvaru <*adresa*> <*příkaz*> <*data*>, kde adresa tentokrát určuje síť z místa *nets*, kam se bude daná zpráva zasílat.

Referenční sítě pro PNVM se v současné době tvoří pomocí nástroje Renew, což je jednak prostředí pro vývoj referenčních sítí, ale také simulátor Petriho sítí. Renew vytvoří síť ve formátu XML, který je třeba převést do jazyka PNAL nebo PNBC[1], což je zjednodušený jazyk vytvořený kvůli zkrácení zápisu sítě, protože celý tento zápis se do PNVM nahraje příkazem *load*, který nahraje danou síť do místa *nets*. K překladu z formátu vytvořený nástrojem Renew se do PNBC převede *translatorem*, což je program specificky vytvořený pro tyto účely. Aktuálně je *translator* ve vývoji, ale to je mimo rámec této práce.

[1] PNAL je to stejné jako PNBC. PNAL je podle [7] a PNBC (Petri Net Byte Code) podle [6]

Následuje ukázka části kódu hlavní sítě v PNBC:

```
(N platform
    ("pn_node_name",
    "setaddr", "pass", "load", "activate", "dump", "loadinst", "unload",
    "delete", "ok", "failed", "bad_cmd", "platform", {"*"}, "*", "&", ".",
    "addroute", "removeroute")

    (address, nets, message, output1, net, output2, output3, routerInput,
    routingTable, routerBuffer)
```

Síť se skládá z elementů ohraničenými závorkami. První element, který zabaluje všechny ostatní, je označen písmenem N a označuje definici šablony sítě, tento prvek je kořenový a může se vyskytnout pouze jednou. Následuje seznam symbolů a seznam míst. Seznam symbolů obsahuje název samotného PNVM jako první symbol a další můžou být pro názvy přechodů nebo jiné textové konstanty, na které se dá odkazovat z dalších elementů definice šablony sítě. Mezi tyto elementy patří přechody, uplinky a inicializační přechod. Jsou to elementy, které mohou být potomky kořenového prvku. Podrobnější popis elementů a celkově PNBC lze nalézt v [7].

Ze seznamu symbolů a seznamu míst je již patrný rozdíl mezi původní hlavní sítí na obrázku 3.2 a současným stavem. Nebudu rozebírat všechny změny, tou nejdůležitější je, že se objevily symboly jako *"*", "&", "."*, které jsou důležité pro komunikaci mezi jednotlivými procesy. Procesem se rozumí běh jedné konkrétní sítě běžící v rámci hlavní sítě nebo i samotná hlavní síť. Zpráva zasílaná mezi procesy je vždy seznam ohraničený složenými závorkami. Dříve zmíněný obecný tvar <*adresa*> <*příkaz*> <*data*> je rozdělen a rozšířen na prvním místě o určení směru zprávy:

- "*" - zpráva je výstupní, síť odesílá zprávu ven.
- "." - zpráva je vstupní, je adresovaná určité síti a pouze ta ji může přijmout.
- "&" - zpráva je žádost o spuštění externího skriptu, tuto zprávu zpracuje Python wrapper, o kterém bude řeč později v kapitole 3.4, spustí skript, například pro čtení dat ze senzoru, a vygeneruje zprávu vstupní s daty získanými ze skriptu.

Celá zpráva je rozdělena opět na tři části jako obecný tvar, jen složení jednotlivých části je jiné. První část je určení směru, další část je seznam obsahující adresu PNVM a určení cílové sítě a příkazu nebo jen příkazu, pokud se posílá hlavní síti, a poslední část jsou data, která bude síť zpracovávat.

```
{<směr>, {<adresa>, <příkaz>}, {<data>}} //zpráva pro nebo od hlavní sítě
{<směr>, {<adresa>, <síť>, <příkaz>}, {<data>}}
```

Pro správný chod PNVM jsou důležité následující příkazy hlavní sítě:

- "load" - nahrání sítě do PNVM, data přijme formátu PNBC
- "activate" - vytvoř instanci nahrané sítě, data jsou typicky seznam s prvky: jméno šablony, jméno instance, inicializační parametry pro danou síť
- "delete" - smaž instanci sítě vytvořené příkazem "activate"
- "addroute" - přidat mapování výstupní zprávy na vstupní zprávu jiné sítě, která může být i v jiném PNVM. Tato funkce umožňuje dynamické směrování mezi sítěmi, lze jej kdykoliv přidat a odebrat
- "removeroute" - odstranit mapování směrování

3.3 PNVM a IoT

V kapitole 2.1 jsem popisoval řešení chytrého domu v podobě softwarových řešení Domoticz a OpenHAB. I do těchto řešení by se dalo zapojit PNVM a tím nahradit logickou část řízení, ale již jsem zmínil, že ani Domoticz ani OpenHAB nejsou úplně ideální pro kombinaci s principy PNVM a možnostmi komunikace přes MQTT. Proto jsem se rozhodl prozkoumat možnosti složení systému chytrého domu s uzly na bázi PNVM s komunikací pomocí MQTT. Součástí systému není mnoho, skládá ze z PNVM uzlů (s python wrapperem pro MQTT), MQTT broker, dashboard pro možnost vidět hodnoty senzorů a ovládat aktuátor a eventuelně i konfigurovat samotné PNVM uzly a nakonec program, který se stará o konfiguraci PNVM uzlů i dashboardu a další potřebné i podpůrné operace pro správný chod systému.Na průzkum těchto věcí se zaměříme v kapitole 5.

3.4 Python wrapper

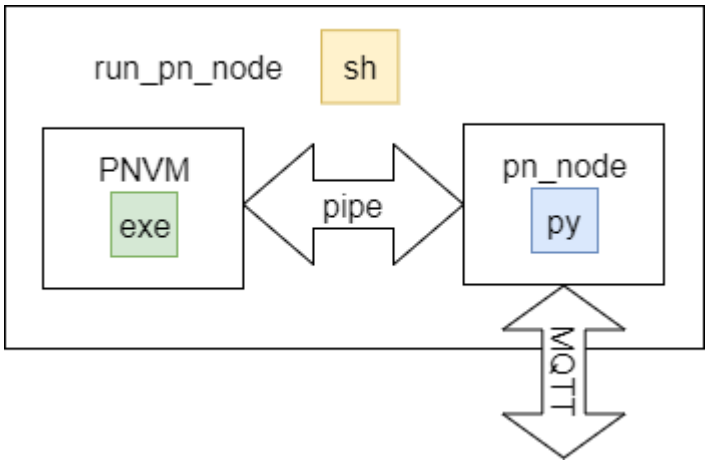
Pro zjednodušení práce s PNVM vznikl jednoduchý skript v jazyce Python. V rámci této práce byl tento skript přepsán pro podporu MQTT, to znamená, že PNVM stále komunikuje přes linux *pipes* svými zprávami, ale jednotlivé instance PNVM, které mohou být i na různých zařízeních v rámci sítě, mezi sebou komunikují pomocí MQTT protokolu.

Celková struktura všeho, co je potřeba k běhu PNVM je následující:

```
root
    conf
    PNOS
        pythonLib
        scripts
        templates
        pn_node
        pnvmx86
        run_pn_node
```

Tato struktura není kompletní, uvádím zde pouze důležité složky a soubory relevantní pro běh samotného PNVM. Složka *conf* na řádku číslo 2 bude diskutována v pozdějších kapitolách. Složka PNOS obsahuje:

- pythonLib - složka s pomocnými funkcemi pro python wrapper
- scripts - složka se skripty pro získávání dat ze senzorů
- templates - složka se šablonami sítí pro PNVM včetně hlavní sítě
- pn_node - samotný python wrapper pro PNVM
- pnvmx86 - spustitelný soubor PNVM
- run_pn_node - bash skript spouštějící PNVM, odtud se spouští i pn_node a pnvmx86. Prvně spustí PNVM a nahraje na něj hlavní síť a poté spustí pn_node pro komunikaci.



Obrázek 3.3: Současný "runtime" PNVM

Z obrázku je patrná komunikace mezi jednotlivými programy. Blíže se budeme věnovat skriptu pn_node, jehož funkcionalitu lze shrnout do následujícího pseudokódu:

```
procedure on_mqtt_message(msg):
    pnvm_msg = transform_mqtt_msg(msg)
    send_to_pnvm(pnvm_msg)

initialize_mqtt(on_mqtt_message)

while 1:
    inputs = [ stdin, async_queue, pipe_in ]
    input, data = read_inputs(inputs)
    if (input == stdin):
        if "&" in data:
            start_async_proccess(data)
        else:
            mqtt_msg = transform_pnvm_msg(data)
            send_mqtt(mqtt_msg)
    else if (input == async_queue): //async_process done
        mqtt_msg = transform_pnvm_msg(data)
        send_mqtt(mqtt_msg)
    else if (input == pipe_in):
        send_to_pnvm(data)
```

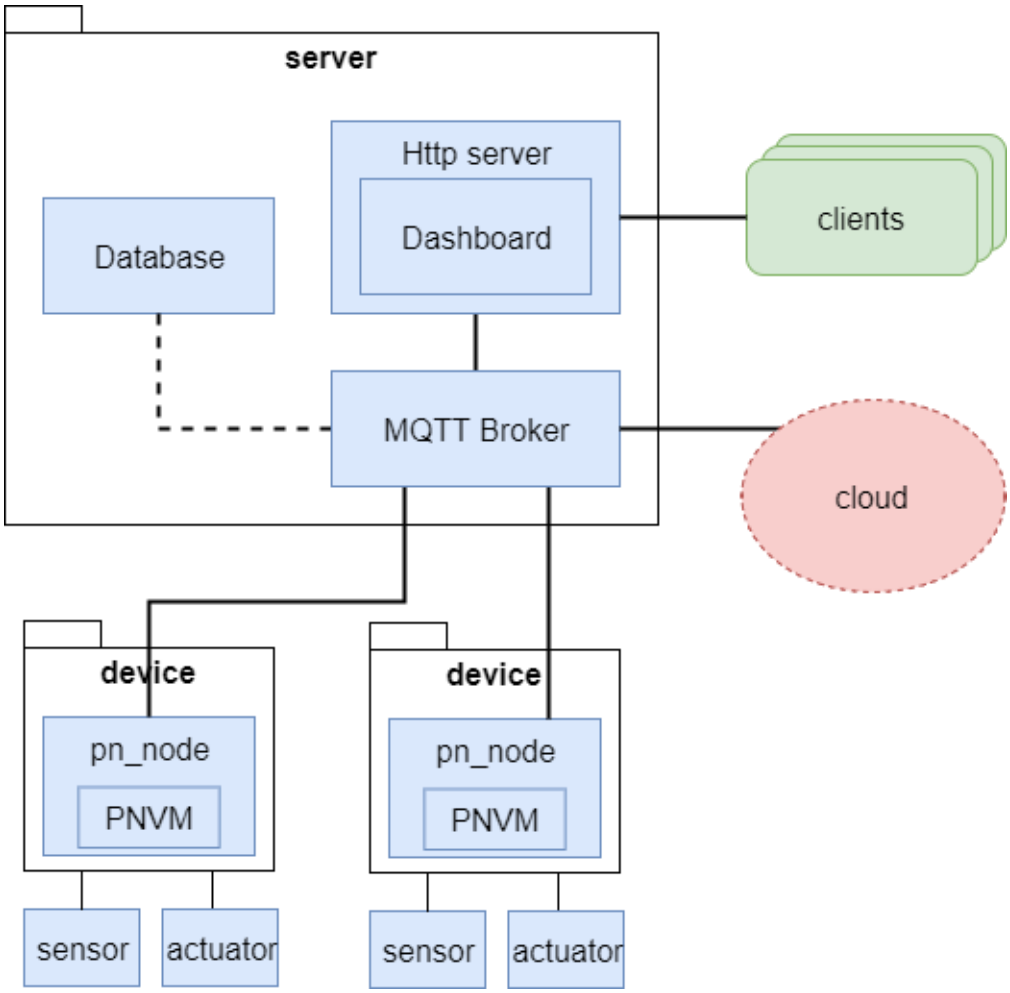
V programu se vyskytuje transformace MQTT zpráv a PNVM zpráv a to oboustranně, té se budu podrobně věnovat v další kapitole. Program tedy po inicializaci MQTT zpracovává vstupy ze čtyř možných vstupů:

- MQTT - zpráva přišla zvenku, poslat ji mohl kdokoliv. V tuto chvíli je jisté, že je určená pro daný PNVM, protože pn_node se přihlašuje k odběru zpráv na endpointu určeném pouze jemu podle jména
- *stdin* - standardní vstup pro pn_node je vstup, kam přicházejí zprávy z PNVM, a standardní výstup pn_node je zase napojen na vstup PNVM a to vše pomocí *pipes*. Zde se ještě rozlišuje, zda si PNVM nežádá o spuštění externího skriptu. V takovém případě pn_node spustí asynchronní proces s vyhodnocením skriptu

- *async_queue* - fronta obsahující asynchronně přicházející výsledky paralelně běžících externích skriptů spuštěných z pn_node na žádost PNVM. Typicky si PNVM vyžádá skript a výsledky chce pro sebe, ale pn_node pošle zprávu přes MQTT jednak z důvodu ladění a také není vyloučeno, že data z externího procesu by mohlo být pro někoho jiného než žadatele.
- *pipe_in* - linuxová *pipe* používající se pro asynchronní vstupy z trvale běžících procesů startovaných z pn_node při jeho startu. Typicky jde o skripty, které okamžitě reagují na přerušení od senzoru, zformulují zprávu pro PNVM, a pošlou ji na příslušný vstup. Z důvodu rychlosti reakce se nejde přes MQTT, i když je to v principu možné, ale kde nevadí drobné zpoždění, je využit MQTT kvůli možnosti sledování komunikace všech komponent.

4 Návrh integrace PNVM do IoT

Středem všeho bude samozřejmě MQTT broker, který vše spojuje. S tím souvisí potřeba integrovat PNVM zprávy, které jsou v tvaru popsaném v kapitole 3.2, aby se dobře posílaly přes MQTT. Další věc je dashboard, který zobrazí stav senzorů, ale i ovládání aktuátorů. Dashboard jako takový bude webová aplikace, která bude nasazená v nějakém webserveru. Základní myšlenka celkové architektury je zobrazena na následujícím obrázku.



Obrázek 4.1: Základní myšlenka architektury

Databáze je úmyslně spojena přerušovaně s MQTT brokerem, protože toto spojení funguje pouze pro ukládání dat. Čtení dat by mělo probíhat z dasboardu, kde se budou data zobrazovat, a přes MQTT broker v tomto případě cesta nevede. Data z databáze budou použity k tvoření grafů, toto ale není zahrnuté v obrázku výše, databáze je v tuto chvíli pouze pro sběr dat.

Z obrázku je patrné, že k serveru může být připojeno více zařízení, které ani nemusí být ve stejné síti. Dále je zde zobrazen i potenciál připojení ke cloudu, kde může být například další MQTT broker vyšší úrovně. Důvod přidání dalšího brokeru spočívá především v tom, že pokud data ze všech zařízení putují přes jeden broker, je tento broker relativně dost vytížený, vzhledem k tomu, že senzory mohou produkovat velké množství dat během krátké doby. MQTT broker na vyšší úrovni by tedy bral jen ta data, která ho zajímají.

4.1 Komunikace PNVM a MQTT

Vzhledem k tomu, že MQTT používá k adresování *topicy*, na které zařízení odesílá zprávy nebo ze které zprávy odebírá, hodí se převádět adresní část z PNVM zprávy do *topicu* a zbytek poslat jako data. Jak bylo zmíněno v kapitole 3.4, python wrapper má v sobě funkce pro převod zpráv, které převádějí zprávy od PNVM do zpráv vhodných k posílání přes MQTT a naopak. Pro jednoduchost budou ty zprávy dále zmiňovány jako PNVM zprávy a MQTT zprávy.

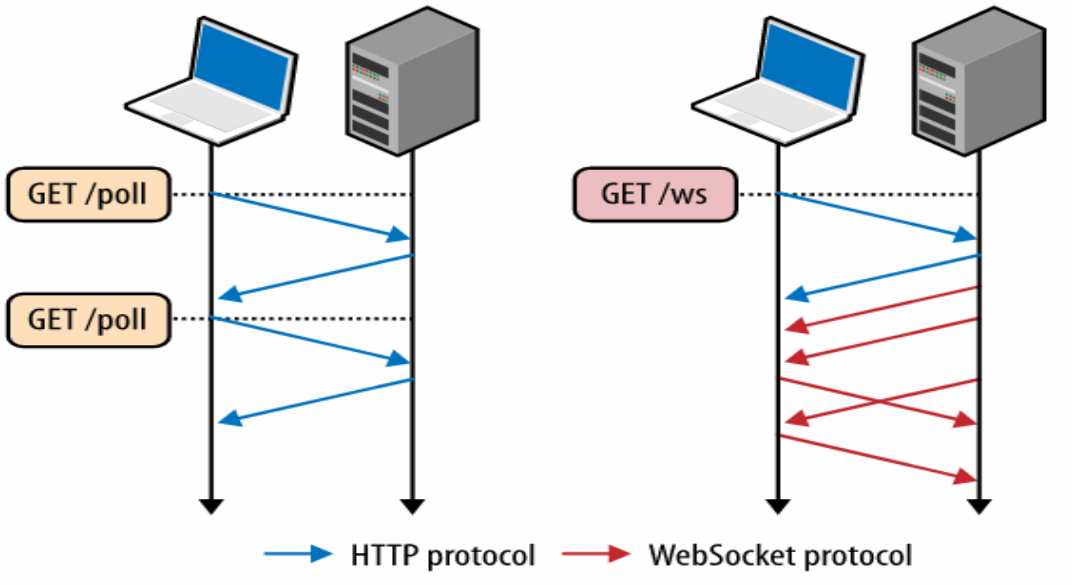
Obecná PNVM zpráva {<*směr*>, {<*adresa*>, <*síť*>, <*příkaz*>}, {<*data*>}} se tedy bude převádět tak, že pouze datová část se bude posílat skutečně jako data a zbytek se převede do *topicu*. Směry lze brát v úvahu pouze dva - vstup a výstup. "&" tedy lze vynechat, neboť jde pouze o požadavek od PNVM, který se ve výsledku převede na vstupní PNVM zprávu.

```
//transformace vstupní zprávy:
{".", {<adresa>, <síť>, <příkaz>}, {<data>}} ->
    topic: in/<adresa>/<síť>/<příkaz>
    data:  {<data>} //včetně složených závorek

//transformace výstupní zprávy:
{"*", {<adresa>, <síť>, <příkaz>}, {<data>}} ->
    topic: out/<adresa>/<síť>/<příkaz>
    data:  {<data>}
```

4.2 Dashboard

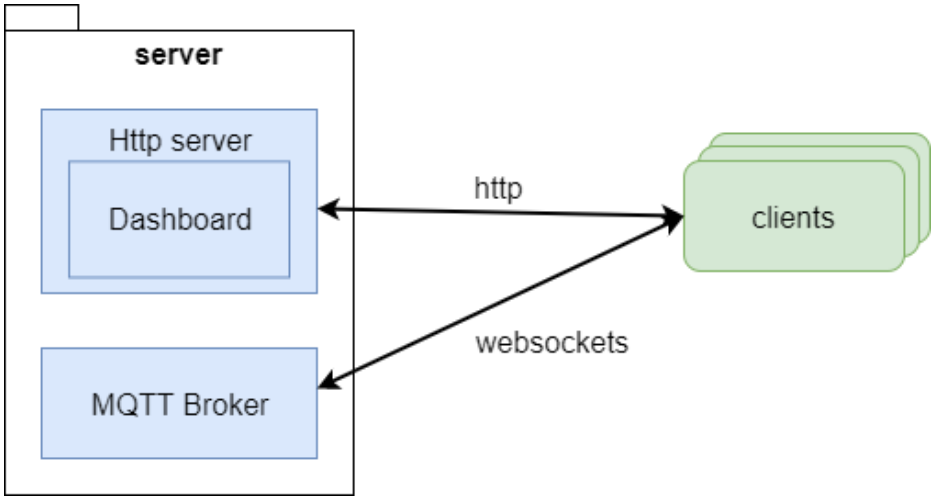
Dashboard bude komunikovat pouze MQTT zprávami, to znamená, že bude nezávislý na použití PNVM. Bude použita technologie *websocketů*, protože dashboard bude webová aplikace, a požadujeme, aby se data projevila okamžitě, což samotný bezstavový http protokol neumožňuje.



Obrázek 4.2: Rozdíl mezi HTTP a websocket protokolem[1]

[1] upravený obrázek z https://hpbn.co/websocket/

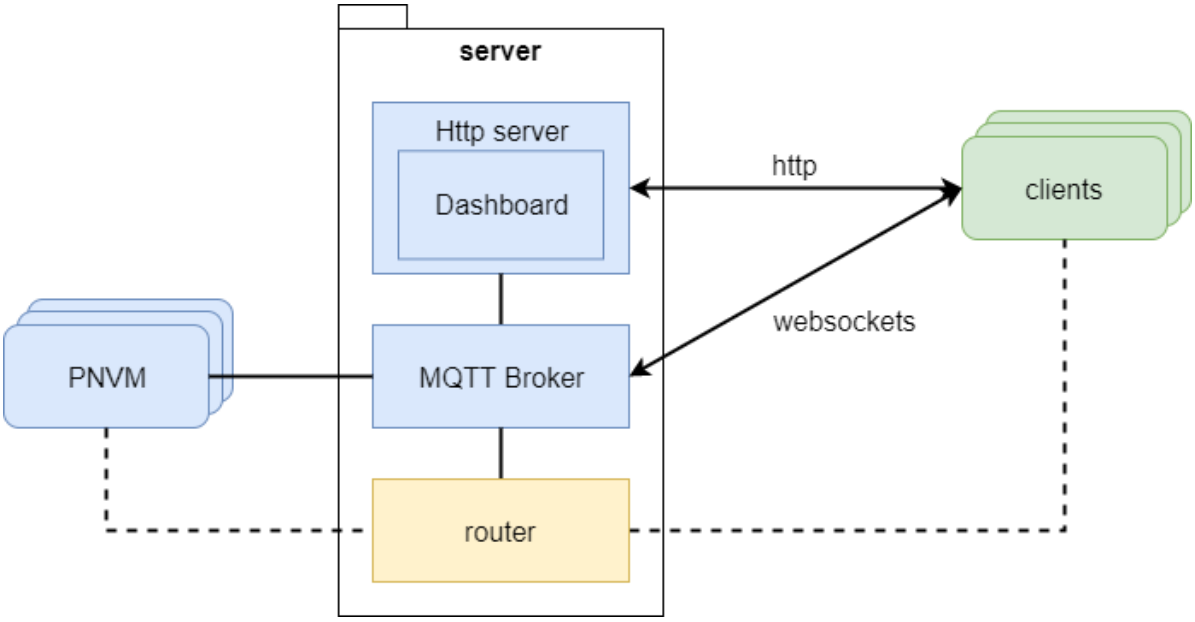
Hlavním úkolem dashboardu bude zobrazit aktuální data ze senzorů a umožnit ovládat aktuátory. Upravená část architektury se zakomponovanými websockety v integraci PNVM do IoT je zobrazena na následujícím obrázku.



Obrázek 4.3: Architektura dashboardu a MQTT s websockety

Také v obrázku chybí přímé spojení mezi MQTT brokerem a dashboardem na serveru. Každý klient má svůj dashboard ve webovém prohlížeči a odtud se přes javascript připojuje k MQTT brokeru websocketovým protokolem. Ideální by bylo, pokud by MQTT broker dokázal obsluhovat jak http část komunikace tak websocketovou. Toto některé MQTT brokery podporují, ale ne na takové úrovni, aby to bylo dobře rozšiřitelné a škálovatelné.

Velmi důležitý je fakt, že pro dashboard bude vybrané open-sourcové řešení, to znamená, že je třeba počítat s tím, že nastane určitá nekompatibilita mezi *topicy*, které bude vyžadovat dashboard, a *topicy*, které používá PNVM (přesněji python wrapper). Z tohoto důvodu se vytvoří python modul *router*, který bude odebírat definované *topicy* a překládat je na jiné, aby si mezi sebou PNVM a dashboard rozuměli.



Obrázek 4.4: Vztah routeru, PNVM a dashboardu (přerušovaná spojení značí účel routeru spojit PNVM a dashboard, ale reálně je router spojen jen s MQTT brokerem)

Další funkce, kterou bude dashboard zastávat, je vývoj samotných Petriho sítí pro PNVM. Aktuálně je možné jednoduše zaslat síť v PNBC přes MQTT, cílem ale je vytvořit Petriho síť pro PNVM v Renew nebo podobném nástroji a výsledek zaslat do PNVM. Zde hraje roli *translator* zmíněný v kapitole 3.2, který překládá sítě z Renew do PNBC. Postup tvorby sítě a následné nahrání je následující:

1. Vytvořit síť v Renew nebo podobném nástroji
2. *translator* přeloží síť do PNBC
3. dojde k odeslání do PNVM
4. instanciace sítě v PNVM nebo jiný příkaz pro PNVM

4.3 Bezpečnost a role

Dashboard přístupný přes webové rozhraní bude potenciálně přístupný každému uživateli internetu. Je tedy třeba řešit bezpečnost, aby chytrý dům mohl ovládat pouze pověřený uživatel. MQTT podporuje možnost přístupu k *topicům* pouze pro specifikované uživatele různými způsoby. Tímto se vyřeší i role, protože určití uživatelé budou mít přistup k jedné množině *topiců* a jiní uživatelé jiné množině *topiců*, přičemž jedna množina může zpřístupňovat pouze zobrazení dat a druhá i ovládání aktuátorů.

Konkrétní role se mohou měnit, ale co týče PNVM, budou to tyto tři:

- uživatel - může vidět data ze senzorů, případně stav aktuátorů
- administrátor - může to stejné co uživatel a navíc ovládat aktuátory
- vývojář - může konfigurovat konkrétní PNVM

Jednotlivé role se mohou prolínat, například administrátor může být zároveň vývojář a podobně.

4.4 Konfigurace a běh PNVM

V tuto chvíli vzhledem k tomu, co v této práci zaznělo, se PNVM spustí se svým python wrapperem nahraje se hlavní síť a zbytek je na uživateli. Cílem této kapitoly je vyřešit konfiguraci při startu i restartu PNVM. Zde přichází na scénu složka *conf* zmíněna na začátku kapitoly 3.4. Ve skutečnosti lze složku PNOS používat samostatně na různých strojích a stejně tak složka *conf* není závislá na přítomnosti složky PNOS.

Tuto složku bude využívat několik python modulů z celkových pěti, které vzniknou pro bezproblémový běh PNVM:

- Supervisor
- Router
- Cache
- Watchdog
- Historian

Modul *supervisor* bude odebírat určitý *topic*, na který bude právě se spouštějící PNVM posílat zprávu o tom, že se připojuje. *Supervisor* přijme tuto zprávu, extrahuje z ní jméno spouštějícího se PNVM a nakonfiguruje ho ze složky *conf*.

Konfigurace je ve formátu *json*, který obsahuje seznam příkazů, které budou poslány přes MQTT k cílovému PNVM. Jeden příkaz se skládá z názvu příkazu pro PNVM a ze samotných dat ve stejném tvaru jako mají data v PNVM zprávě.

```
[ "<příkaz>", "{<data>}" ] //obecně jeden příkaz

//konkrétní příklad s nahráním sítě a vytvoření instance sítě
[
    [ "load", "{<PNBC kód sítě>}" ],
    [ "activate", "{\"sit\", \"nazev_instance\", \"param1\", \"param2\"}" ]
]
```

Podobně bude *supervisor* konfigurovat i dashboard, pokud bude dashboard vyžadovat externí konfiguraci.

Modul *router* byl již vysvětlen v kapitole 4.2, takže se lze přesunout rovnou na modul *cache*. Ten se bude starat o ukládání hodnot z určených *topiců* z konfigurace na souborový systém. Účelem je nastavení hodnot, které se posílaly naposled, po restartu serveru nebo při novém přístupu na dashboard. K mapování se využije toho, že *topic* má charakter systémové cesty. Například:

```
topic: out/instance/status
msg: {200}
```

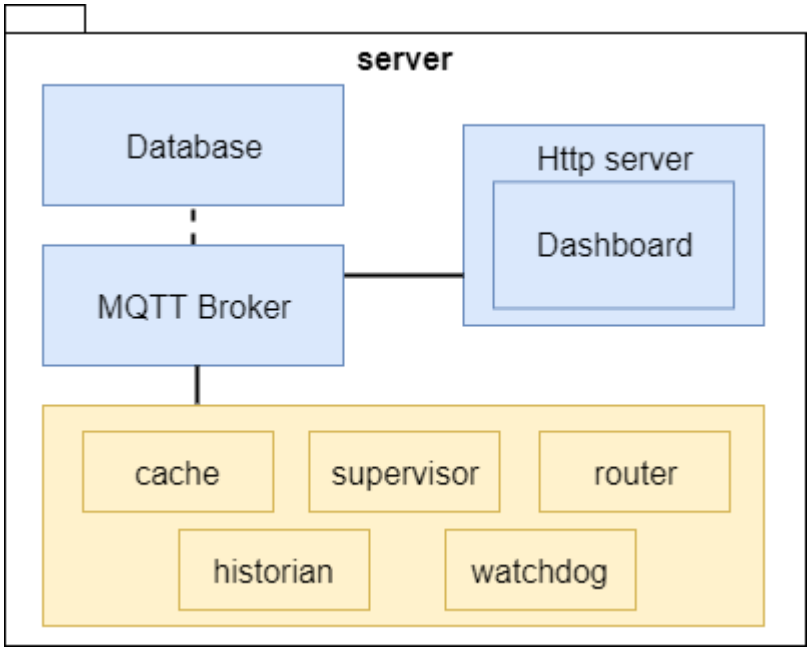
Se namapuje do systému následovně:

```
složka: /cache_slozka/out/instance/
soubor: status
obsah souboru: {200}
```

To znamená, že celý *topic,* kromě posledního prvku bude cesta, poslední prvek bude soubor a data budou obsahem souboru. Tím se zajistí, že se uchová pouze ta nejnovější hodnota, která je pro tento případ nejdůležitější. Samotný obsah cache poté využije *supervisor* pro konfiguraci PNVM nebo dashboardu, kdy po nakonfigurování ještě pošle hodnoty z cache.

Modul *watchdog* má za úkol hlídat, zda všechny procesy PNVM probíhají jak mají. Může se jednat o jednoduché sledování *topicu*, a pokud se na něm určitý čas nic neobjeví, znamená to, že například senzor přestal posílat data. Na druhou stranu, lze komplexně kontrolovat stav celého PNVM z jeho *dumpu*, tj. z výsledku po příkazu *dump,* jehož výsledkem je PNBC. V *dumpu* se totiž nachází stav všech sítí a jejich instancí v PNVM. *Watchdog* je konfigurovaný na sledování specifických *topiců* z konfiguračního souboru ve složce *conf*.

A nakonec modul *historian* nedělá nic jiného, než že ukládá vybrané *topicy* a zprávy do databáze. Pokud ve zprávě detekuje číselné hodnoty, uloží je jako číselnou hodnotu pro možnost pozdějšího využití při tvorbě grafů.



Obrázek 4.6: Programy běžící na serveru

5 Software pro integraci PNVM do IoT

V této kapitole najdeme vhodné prvky pro kombinaci s PNVM a MQTT. Zaměříme se na výběr konkrétního MQTT brokeru, databáze a dashboardu.

5.1 MQTT broker

Co se týče MQTT brokeru, není třeba speciálně vybírat, MQTT specifikaci musí splňovat všechny implementace, a to je to hlavní co nás zajímá. Proto jsem ani nijak neváhal a rovnou jsem šáhnul po Mosquitto brokeru, což je open-sourcové řešení s velkou komunitou a velmi rozsáhlou, přehlednou a jasnou dokumentací dostupnou na [8].

5.2 Databáze

Výběr databáze už je složitější, neboť zde vyvstává otázka, zda bude jedna velké databáze na serveru s brokerem nebo na dedikovaném serveru někde v cloudu a nebo bude databáze na každém uzlu nebo obojí. Podpora databáze a MQTT není nutná přímo, protože nám stačí, aby se dalo k databázi přistoupit z jazyka Python, ve kterém je napsaný wrapper pro PNVM.

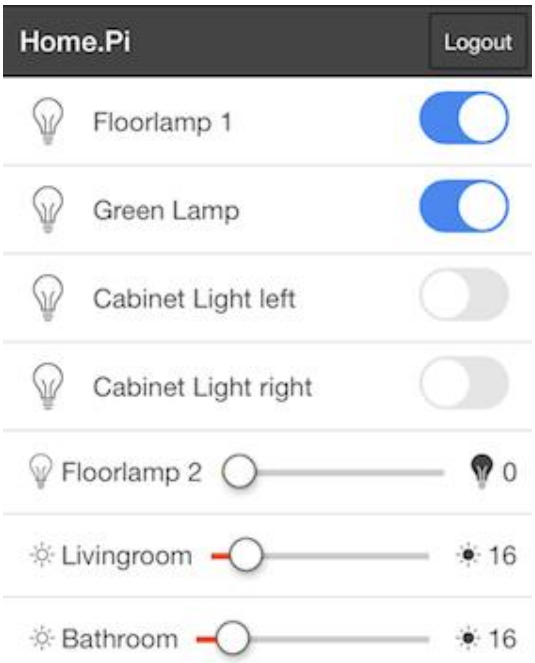
Jako první se nabízí klasické relační databáze jako lehké *SQLite* nebo více komplexní jako *PostgreSQL*. Vše to jsou databáze s dobrou podporou a dokumentací a všechny mají knihovny pro přístup z jazyka Python. Poté tu máme NoSQL databáze jako například *CouchDB*, což je databáze, která se hodí jak pro big data tak pro malá zařízení.

Speciálním druhem databází, který se hodí především v oblasti IoT na čtení velkého množství dat ze senzorů jsou databáze časových sérií. Mezi ně patří například *RRDTool*, která je velmi výkonná a dokáže nejen ukládat časová data, ale i z nich tvořit grafy. Tato databáze je unikátní v tom, že data ukládá cyklicky, to znamená, že pro data alokuje přesně vymezený prostor daný časem, a jakmile dojde, začne přepisovat hodnoty od začátku. Další takovou databází je *InfluxDB*. Ta je navržena tak, aby zvládala velké přílivy dat najednou. Také má http API pro správu databáze, takže se lehce zkombinuje s MQTT a Pythonem, což z ní dělá ideálního kandidáta k použití.

5.3 Dashboard

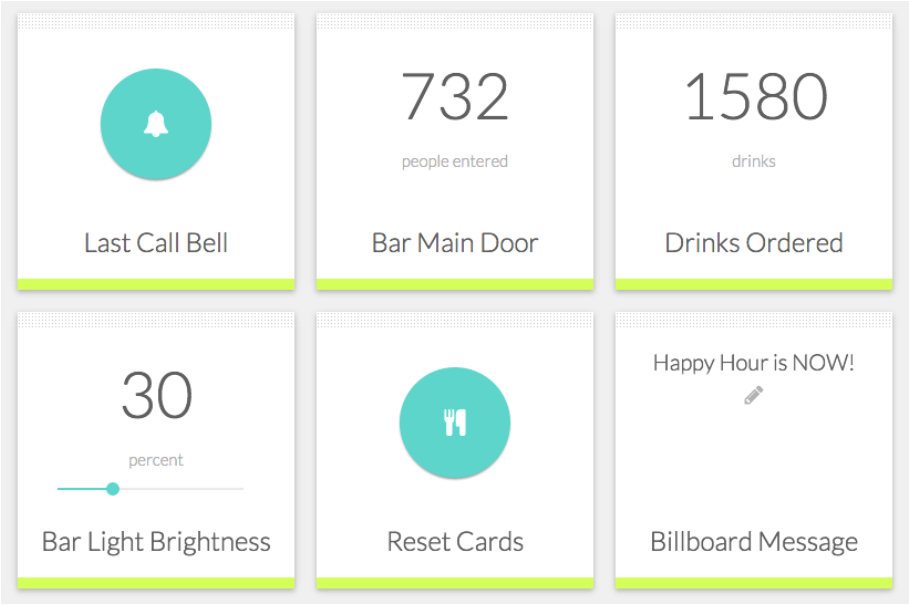
Dashboardů s podporou MQTT existuje spousta. Narazil jsem ale na několik, které se přímo hodí k integraci s PNVM.

Jako první je to *home.pi*, který je implementovaný v javascriptu. Je jednoduchý a optimalizovaný pro zobrazení na mobilních zařízeních. Jeho nastavení probíhá zasláním jednoho velkého jsonu na specifický topic. V tomto jsonu jsou definované všechny prvky chytrého domu.



Obrázek 4.1: Ukázka home.pi rozhraní[1]

Dalším dashboardem je *Crouton* také implementovaný v javascriptu. Funguje podobně jako home.pi, ale konfigurace je propracovanější. Každé zařízení posílá inicializační MQTT zprávu v json tvaru, která je pro zařízení specifická a definuje všechny jeho senzory a aktuátory. Tento způsob je přesně to, co chceme v souvislosti s PNVM a dynamickou rekonfigurací. Pokud by se do tohoto jsonu přidala definice referenční sítě pro PNVM, byl by to jediný a kompletní konfigurační soubor pro definici celého jednoho PNVM uzlu, to se ale ukázalo nepřesné, protože dashboard bude jen jeden a PNVM uzlů může být více.



Obrázek 4.2: Ukázka Crouton dashboardu[2]

[1] https://github.com/denschu/home.pi/blob/master/screenshot.png?raw=true
[2] http://crouton.mybluemix.net/static/common/images/crouton-howto-4.png

6 Demonstrační implementace

Demonstrační aplikace bude spočívat v ovládání teploty a vytápění ve dvou místnostech. To znamená dva teploměry, dva termostaty a boiler. Vše poběží na jedné instanci PNVM, ale nebylo by těžké modifikovat příklad pro více instancí. Bude použit modifikovaný dashboard Crouton, webserver *lighttpd* a testovací prostředí bude Raspberry Pi 2.Logická část se bude skládat z několika částí:

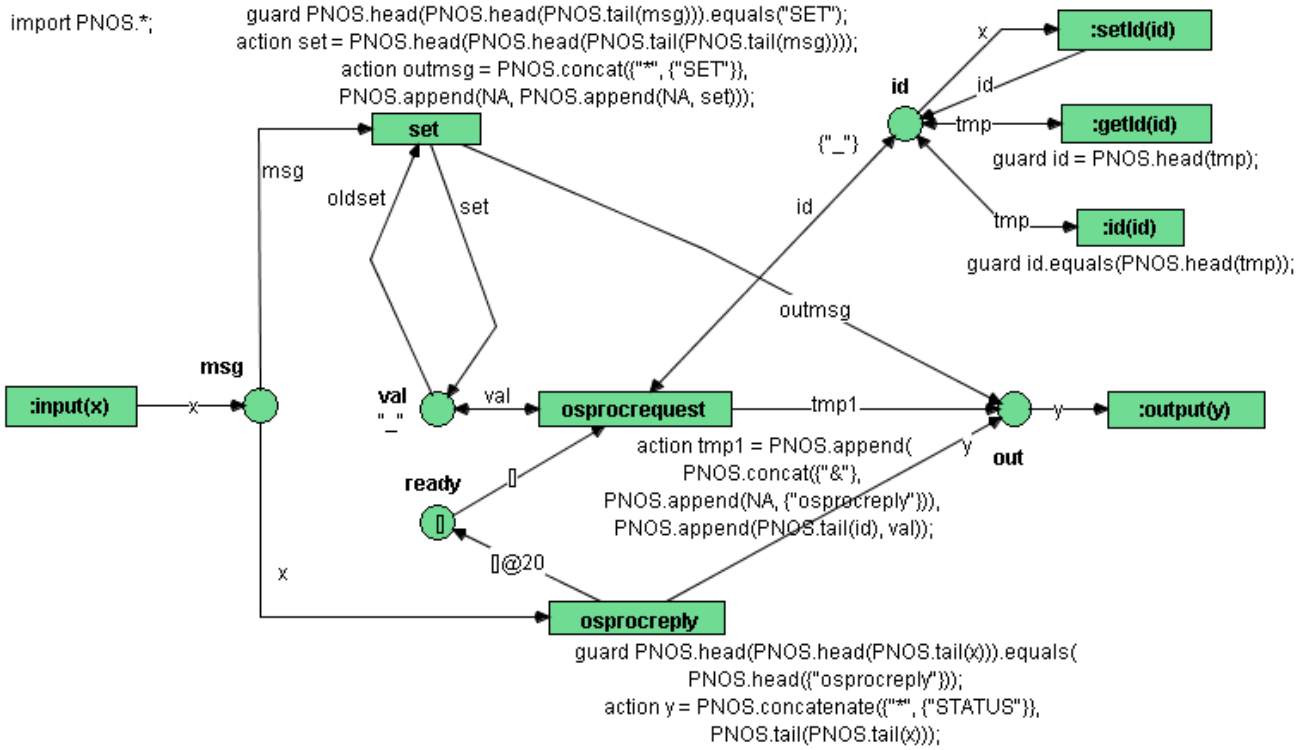
- teploměr pro měření teploty v místonosti
- termostat pro určení kdy otevřít radiátor
- radiátor
- "heating control' komponenta, která zapne boiler na základě stavu radiátorů. To znamená, že boiler se zapne, když je alespoň jeden radiátor otevřený
- boiler

6.1 Konfigurace

6.1.1 PNVM

První je konfigurace PNVM, o které už byla řeč v kapitole 4.4. Nutno dodat, že pro příkaz "load" je definována zkratka "loadTemplate", která umožní nezadávat celý PNBC do konfiguračního souboru, ale mít šablony sítí v jiné složce. Na tento příklad budou použity celkem tři Petriho sítě:

- SensorPoll_20s - nemá žádný vstup a má jeden výstup. Každých 20 sekund přečte data ze senzoru a vrátí výsledek
- ActPoll_20s - má jeden vstup a jeden výstup. Každých 20 sekund zapíše aktuální stav do aktuátoru, který má nastaven na vstupu
- Controller - má dva vstupy a jeden výstup. Jeden vstup funguje jako setpoint, který nastaví číslo pro porovnávání. Druhý vstup očekává opět číslo, a když dojde, porovná se s prvním a podle výsledku na výstup pošle buď "1" nebo "0", podle toho, jestli bylo číslo menší respektive větší než setpoint



Obrázek 6.1: Ukázka sítě ActPoll_20s

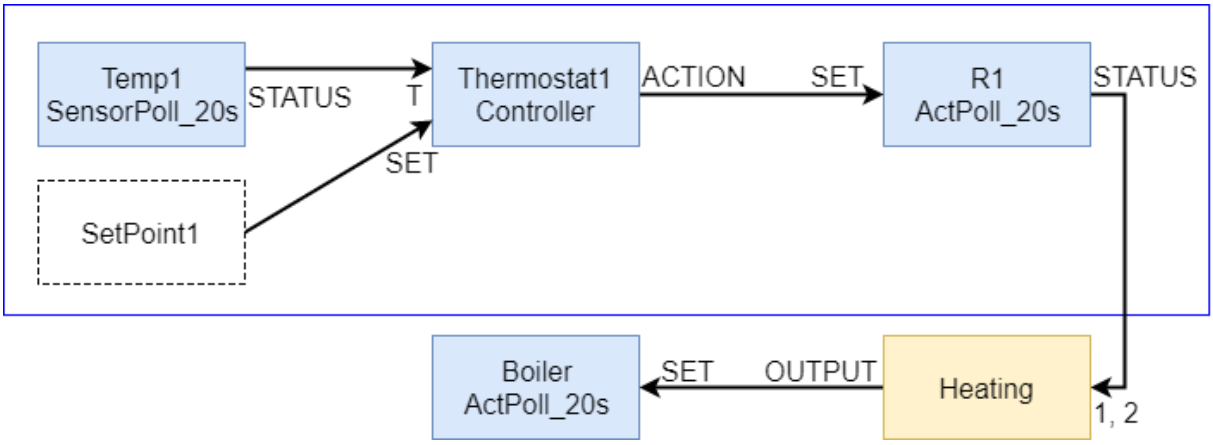
```
[
  [ "loadTemplate", "SensorPoll_20s" ],
  [ "loadTemplate", "ActPoll_20s" ],
  [ "loadTemplate", "Controller" ],

  [ "activate", "{\"SensorPoll_20s\", \"Temp1\", \"DomoticzRead\",
    \"Temp1\"}" ],

  [ "activate", "{\"Controller\", \"Thermostat1\", 220}" ],

  [ "addroute",
"{{{\"#ADDR\",\"Temp1\",\"STATUS\"},{{\"#ADDR\",\"Thermostat1\",\"T\"}}}}"
  ]
]
```

Ukázka konfiguračního souboru pro PNVM, konkrétně je zde ukázána aktivace teploměru a termostatu s nastavením výstupu teploměru na vstup termostatu, který je inicializován na teplotu 22 °C. Konstanta *"#ADDR"* bude nahrazena jménem PNVM instance, což je v tomto případě *pnos-test*. Kompletní struktura sítí a jejich propojení je znázorněna na následujícím obrázku.



Obrázek 6.2: Struktura PNVM sítí demonstračního příkladu

Modré rámečky označují sítě, kde první řádek je název instance a druhý název šablony sítě. Bíle označený *SetPoint* není vůbec síť, ale bude se nastavovat přímo z dashboardu. V modrém rámečku jsou sítě pro konkrétní pokoj, v demonstračním příkladu tedy bude existovat každá z nich dvakrát. Komponenta "Heating" má dva vstupy, pro radiátor z každého pokoje. Oranžovou barvou je označen z toho důvodu, že nebude nahrán do PNVM jako Petriho síť, ale bude to externí nezávislý python skript, který bude napojen přímo na MQTT. Tím se dokáže, že PNVM je vysoce flexibilní a není nutné dělat vše pomocí Petriho sítí, obzvláště pokud jde například o existující řešení, které stačí napojit na MQTT.

6.1.2 Cache

Dále je zde konfigurace cache, která opět ve formátu *json* vyjmenovává pole *topiců*, které budou cachovány i s daty.

```
[
  "pnos/out/pnos-test/Temp1/STATUS",

  "pnos/in/pnos-test/Thermostat1/T",
  "pnos/in/pnos-test/Thermostat1/SET",
  "pnos/out/pnos-test/Thermostat1/SETPOINT"
]
```

V ukázce kódu je poprvé vidět, že *topic* týkající se PNVM bude vždy začínat předponou *pnos*. Je to také definice pro stejnou část příkladu jako v případě konfigurace pro samotné PNVM, tedy první teploměr a první termostat.

6.1.3 Dashboard

Následuje konfigurace dashboardu. Crouton má svoji specifickou konfiguraci, která se také provádí pomocí souboru ve formátu *json*. Podobně jako funguje nastavování PNVM *supervisorem*, funguje defaultně nastavování Croutonu. Konfigurační souboru Croutonu vypadá následovně:

```
{
  "deviceInfo": {
    "name": "pnos-test",
    "endPoints": {
      ...
    },
    "description": "PNOS"
}
```

Důležitá je položka "endPoints", kde jsou definovány veškeré prvky dashboardu, konkrétní příklad prvku z demonstračního příkladu pro zobrazení teploty:

```
"Temp1": {
  "title": "Tmp 1",
  "card-type": "crouton-simple-text",
  "units": "C",
  "values": {
    "value": 0
  }
}
```

Pro nastavení termostatu:

```
"Therm1": {
  "title": "Therm 1",
  "card-type": "crouton-simple-input",
  "values": {
    "value": 220
  }
}
```

A pro zapnutí a vypnutí radiátoru:

```
"R1": {
  "title": "R1",
  "card-type": "crouton-simple-toggle",
  "labels": {
    "true": "On",
    "false": "Off"
  },
  "icons": {
    "true": "sun-o",
    "false": "moon-o"
  },
  "values": {
    "value": false
  }
}
```

Název endpointu je důležitý pro správné mapování v MQTT, ostatní hodnoty specifikují vzhled prvku na dashboardu, jako titulek, defaultní hodnota nebo typ karty. Crouton nabízí několik typů základních karet: [15]

- Text - zobrazení jakéhokoliv textu
- Input - zobrazení textu s možností jej změnit. Samozřejmě se tento text pošle i přes MQTT stejně jako u všech aktivních prvků Croutonu
- Slider - zobrazení číselné hodnoty s posuvníkem, kde je možné nastavit minimální a maximální hodnotu, mezi kterými lze posouvat
- Button - klasické tlačítko. Při stisknutí odešle hodnotu "true"

- Toggle - přepínač, posílající "true" nebo "false" podle aktuálního stavu

U všech aktivních prvků Croutonu platí, že ta aktivní část změní stav okamžitě po kliknutí, ale je zde ještě pasivní zobrazovací část, která změní stav až po jeho obdržení od MQTT. Tím pádem lze pozorovat zpoždění, které může v rámci PNVM procesů nastat a které reálně v tomto demonstračním příkladu nastává.

Konfigurace Croutonu obsahuje ještě jednu důležitou definici, a to definici pohledů. Ta není standardní součástí Croutonu, ale vznikla jako modifikace v rámci této práce.

```
"description": "PNOS",
"views": {
  "test": [ "toggle", "slider" ],
  "teplomery": [ "Temp1", "Temp2" ],
  "termostaty": [ "Therm1", "Therm2" ],
  "radiatory_boiler": [ "R1", "R2", "Boiler" ],
  "watchdog_status": [ "Temp1_status", "Temp2_status" ]
}
```

Položka "description" je zde uvedena jen pro představu, kde se definice pohledů nachází v rámci konfiguračního souboru. Samotná definice pohledů je seznam, kde položka se skládá z jména pohledu a výčtu Croutonovských endpointů, které do pohledu patří. Více se budu věnovat pohledům v kapitole 6.3.

6.1.4 Router

Další důležitou konfigurací je ta pro *router*, ta je zde kritická především kvůli dashboardu, protože *topicy,* které používá PNVM se liší od těch používaných Croutonem. Konfigurace pro *router* je opět soubor ve formátu *json* a opět jde o seznam definic pro jednotlivá mapování. Mapování je kardinality 1:N s tím, že mapovaný zdrojový *topic*, může mít více cílů. Navíc je zde ještě uvedená transformace, která se aplikuje na data zdrojového *topicu*, aby se z nich stala data vhodná pro cílové *topicy*. Uvádím zde opět příklad pro první teploměr a termostat, jak se mapují z PNVM do Croutonu:

```
"pnos/out/pnos-test/Temp1/STATUS": {
  "to": [ "/outbox/pnos-device/Temp1" ],
  "transform": "to_outbox"
},

"/inbox/pnos-device/Therm1": {
  "to": [ "pnos/in/pnos-test/Thermostat1/SET" ],
  "transform": "from_inbox"
},
"/out/pnos-test/Thermostat1/SETPOINT": {
  "to": [ "/outbox/pnos-device/Therm1" ],
  "transform": "to_outbox"
}
```

Z ukázky je vidět, jaké *topicy* Crouton používá. Na začátku je vždy buď "outbox" nebo "inbox" podle toho, zda jde o data posílaná do komponent Croutonu ("outbox") nebo data, která posílá Crouton zpět ("inbox"). Transformace je v tomto případě přímo název funkce definované v pythonu. Je to funkce, která přijme MQTT data jako textový řetězec a vrátí opět textový řetězec. Nemusí být ani definována v jiném souboru, lze zapsat přímo "inline" do konfigurace, například:

```
lambda data: data.lower()
```

Je jednoduchá transformace, která převede velká písmena v datech na malá. V demonstračním příkladu jsou ale transformační funkce definované v python souboru, aby se tato definice nemusela opakovat na tolika řádcích v konfiguraci.

6.1.5 Watchdog

Nakonec je zde konfigurace pro *watchdog*, která již standardně je v souboru ve formátu *json* a definuje pole pravidel pro sledování. Pravidlo se skládá z definice *topicu* pro sledování a času v sekundách, který když uběhne bez aktivity na daném *topicu,* zašle *watchdog* zprávu o neaktivitě. Pro tento demonstrační příklad sleduje *watchdog* pouze teploměry a není definována žádná komplexní logika sledování, jde pouze o *timeout*.

```
{
  "pnos/out/pnos-test/Temp1/STATUS": 30,
  "pnos/out/pnos-test/Temp2/STATUS": 30
}
```

A to je kompletní konfigurační soubor pro modul *watchdog*.

6.2 Python moduly

6.2.1 Cache

Cache modul si po spuštění vezme *topicy* z konfigurace a začne je odebírat. Ukládání do souborového systému je nastaveno do složky */tmp/mqtt.*

```
def write_to_system(topic, content):
    path_split = topic.split('/')
    directory = os.path.join(cache_dir, *path_split[:-1])
    file_name = path_split[-1]

    if not os.path.exists(directory):
        os.makedirs(directory)

    with open(os.path.join(directory, file_name), 'w') as file:
        file.write(content)
```

Tato funkce vezme *topic* z příchozí zprávy, rozdělí jej na části tak, že všechny části kromě poslední tvoří cílovou složku a poslední část je název souboru. Nakonec otevře tento soubor pro zapisování a uloží zprávu. Soubor je otevírán v módu zápisu, který vytvoří daný soubor, pokud neexistuje, a nebo soubor přepíše, pokud existuje.

```
def all_from_cache():
  for cur, dirs, files in os.walk(cache_dir):
    for file in files:
      with open(os.path.join(cur, file)) as f:
        yield ('{}/{}'.format(cur[len(cache_dir) + 1:], file), f.read())
```

Funkce *all_from_cache* naopak načte cely obsah */tmp/mqtt* a poskládá opět vše do tvaru *(topic, data)*, což je pythonovská dvojice. Tuto funkci bude používat *supervisor* pro inicializaci stavu PNVM nebo dashboardu.

6.2.2 Historian

Modul historian používá v demonstračním příkladu stejný konfigurační soubor jako cache. Takže stejně jako cache se při spuštění nakonfiguruje pro odběr *topiců* ze seznamu. Historian používá pythonovskou knihovnu pro komunikaci s InfluxDB[1].

```
def mqtt_msg_to_influx_data(msg):
    def msg_to_fields(msg):
        msg_str = str(msg.payload).replace('{', '').replace('}', '')
        try:
            return {"value": float(msg_str)}
        except ValueError:
            return {"stringValue": msg_str}

    return [{
        "measurement": "messages",
        "tags": {"topic": msg.topic},
        "fields": msg_to_fields(msg)
    }]
```

Funkce, která převádí MQTT zprávu do tvaru, který přijme InfluxDB. Prvně odstraní z dat znaky "{" a "}", které jsou v každé zprávě generované PNVM. Pokud to není zpráva od PNVM, nic se neděje, v tom případě znaky neodstraní. Pokud je obsahem dat číslo, bude se ukládat do sloupce *"value"*, pokud ne tak do *"stringValue"*. A na konci je tvar zprávy, kterou očekává InfluxDB. Je to vždy pole a každý prvek očekává následující hodnoty:

- *measurement* - název tabulky do které se zpráva ukládá.
- *tags* - seznam štítků, která definují metadata prvku. Tato data se indexují pro rychlé vyhledávání.
- *fields* - hodnoty skutečných dat. Můžou být textové řetězce, desetinná čísla, celá čísla nebo booleaovské hodnoty. Tato data nejsou indexovaná.
- *timestamp* - časové razítko záznamu. Pokud není uvedeno, je použit aktuální čas.

6.2.3 Router

Router si konfigurací načte směrovací tabulku do lokální proměnné, kde bude ve tvaru klíč-hodnota.

```
def on_message(client, userdata, msg):
    target = routing_table[msg.topic]
    data = str(msg.payload)

    if 'transform' in target:
        data = eval(target['transform'])(data)

    for dest in target['to']:
        print("%s -> %s : %s" % (msg.topic, dest, data))
        client.publish(dest, data)
```

on_message je funkce, která funguje jako *callback* pro přijetí MQTT zprávy. Pokud tato funkce byla zavolána, je jisté, že směrování pro *topic* bude existovat, protože router odebírá pouze ty *topicy*, které mají definované směrování. Router tedy zavolá transformační funkci na daty, pokud tato funkce existuje, a transformovanou zprávu odešle na všechny cílové *topicy*.

[1] https://github.com/influxdata/influxdb-python

6.2.4 Supervisor

Supervisor se stará o konfiguraci PNVM a dashboardu, on sám se nijak nekonfiguruje snad kromě jednoho seznamu *topiců* pro odebírání přímo v jeho kódu, které jsou následující:

- *pnos/out/+/online* - sem přijde zpráva od PNVM, které se právě spouští, což je signál pro odstartování jeho konfigurace.
- */inbox/pnos-device/deviceInfo* - *topic*, na který posílá Crouton požadavek o informace k zobrazovaným prvkům.
- *pnos/out/+/lwt* - MQTT *last will topic*, sem se odešle zpráva, pokud se PNVM nečekaně odpojí.

Modul supervisor musí být spuštěn ještě předtím než se spustí PNVM, aby ho správně nakonfiguroval.

```
def on_message(client, userdata, msg):
    if "deviceInfo" in msg.topic:
        print 'initializing crouton device {}'.format(crouton_device_name)
        send_crouton_conf(client)

    elif "online" in msg.topic:
        addr = str(msg.payload)
        print 'configuring device {}'.format(addr)

        device_conf = get_device_conf(addr)
        for cmd in device_conf:
            if cmd[0] == 'loadTemplate':
                client.publish('{}/in/{}/load'.format(topic, addr),
                               load_template(cmd[1]))
            else:
                client.publish('{}/in/{}/{}'.format(topic, addr, cmd[0]),
                               cmd[1])

        init_from_cache(lambda (topic, msg): addr in topic)

    elif "lwt" in msg.topic:
        addr = msg.topic.split('/')[-2]
        print 'device {} disconnected'.format(addr)
```

Zpracování MQTT zprávy je rozděleno na tři části podle toho, na který *topic* zpráva přišla. Pokud jde o inicializaci Croutonu, supervisor mu obratem pošle jemu určený obsah konfiguračního souboru, který je následován inicializací z cache. Pokud je to zpráva od PNVM o spuštění, supervisor pošle jednotlivé příkazy definované v konfiguračním souboru. Když se jedná o příkaz "loadTemplate", načte prvně PNBC sítě. Jakmile poslal PNVM všechny příkazy a nahrál všechny sítě, dojde k inicializaci z cache, která vypadá následovně:

```
def init_from_cache(filter_fun=None):
    for topic, msg in filter(filter_fun, all_from_cache()):
        client.publish(topic, msg)
```

all_from_cache je funkce definovaná v routeru, která má v sobě dvoje *(topic, data)*. Takže supervisor jednoduše iterativně odešle všechny hodnoty z cache při jakékoliv inicizalizaci. Funkce pro inicializace z cache obsahuje ještě filtr, kterým lze omezit rozsah posílaných hodnot z cache.

6.2.5 Watchdog

Watchdog pracuje se seznamem *topiců*, které odebírá, a pokud nepřijde na daný *topic* zpráva do definovaného časového limitu, odešle zprávu, že proces je offline.

```
class State:
    def __init__(self, timeout, on_timeout):
        self.prev_state = None
        self.timeout = timeout
        self.on_timeout = on_timeout
        self.timer = Timer(timeout, self.reset)
        self.timer.start()

    def reset(self, state='offline'):
        self.cancel()
        self.timer = Timer(self.timeout, self.reset)
        self.timer.start()
        if state != self.prev_state:
            self.on_timeout(state)
        self.prev_state = state

    def cancel(self):
        self.timer.cancel()
```

Watchdog používá třídu *State*, která má parametr pro uchování předchozího stavu, který může nabývat hodnot "online" a "offline", parametr pro časovou prodlevu, *callback* funkci, která se volá při dosažení časové prodlevy ale i při resetu časovače, a samotný časovač. *State* má dvě metody:

- *reset* - resetování časovače, která zároveň volá *callback* funkci, pokud se stav změnil.
- *cancel* - zrušení časovače

```
def on_message(client, userdata, msg):
    timer = watch_table[msg.topic]
    timer.reset('online')

def on_connect(client, userdata, flags, rc):
    client.subscribe(map(lambda topic: (str(topic), 0), watch_table))
    for topic in watch_table:
        timeout = watch_table[topic]
        watch_table[topic] = State(timeout, on_timeout(client, topic))
```

Spuštění funkce *on_message* znamená, že na daném *topicu* je aktivita, takže se časovač resetuje aktuálním stavem "online". Stejně jako router zde watchdog ví, že časovač existuje, protože odebírá pouze definované *topicy*.

Funkce *on_connect* proběhne jednou při spuštění programu a jejím úkolem je načíst konfiguraci modulu watchdog a začít odebírat dané *topicy*. Při inicializaci také změní časovou prodlevu z konfigurace na instanci třídy *State*, čímž rovnou spustí časovač s danou prodlevou.

Callback funkce *on_timeout* nedělá nic jiného, než že odešle na specifický *topic* zprávu "online" nebo "offline" podle stavu. Tento *topic* je téměř stejný jako *topic*, který watchdog sleduje, ale změní poslední složku za *watchdog_status*, takže například *topic*:

```
pnos/out/pnos-test/Temp1/STATUS
```

se změní na:

```
pnos/out/pnos-test/Temp1/watchdog_status
```

6.2.6 Shrnutí

Celkově tedy existuje 5 modulů, které se starají o chod celého systému i s PNVM. Pro jednoduché spouštění vznikne ještě jeden modul s názvem *all_mqtt*, který bude spouštět všechny ostatní moduly a zobrazovat jejich logovací výstup.

```
def create_processes(q):
    names = [
        'cache',
        'historian',
        'router',
        'supervisor',
        'watchdog'
    ]

    processes = []
    for name in names:
        print 'creating {} process'.format(name)
        p = multiprocessing.Process(target=run_command, args=(name, q,))
        processes.append(p)

    return processes
```

Touto funkcí budou vytvořeny procesy pro každý modul a dále v programu paralelně spuštěny a bude se vypisovat jejich výstup.

```
def run_command(cmd, q):
    process = subprocess.Popen(...)
    try:
        while True:
            line = process.stdout.readline()
            if line == '' and process.poll() is not None:
                break
            q.put('{} [{}] >>> {}'.format(datetime.now(), cmd, line))
    except KeyboardInterrupt:
        process.stdout.close()
```

Funkce *run_command* je definicí běhu jednoho modulu. Tato funkce spustí proces a čte výstupy, které proces vyprodukuje. Výstupy jsou ukládány do fronty společné pro všechny procesy, z které poté hlavní funkce programu *all_mqtt* čte a vypisuje výstupy.

Poslední python program, který je součástí demonstračního příkladu, je *demo_heating*. Jde o již zmiňovanou implementaci komponenty "heating control", která nemá svoji Petriho síť, ale je implementována právě zde. Tato komponenta má 2 vstupy a jeden výstup, takže dva *topicy* na odebírání a jeden na posílání:

- *pnos/out/pnos-test/R<i>/STATUS* - vstupní *topicy* z výstupu radiátoru, kde <i> $\in \{1, 2\}$
- *pnos/in/pnos-test/Boiler/SET* - výstupní *topic* je vstupním *topicem* boileru

```
def on_message(client, userdata, msg):
    num = int(msg.topic[-8:-7])
    if state.change(num, str(msg.payload)):
        client.publish(mqtt_response, '{{"{}"}}'.format(str(state.curr)))
```

Při zpracování zprávy jen zjistí, od kterého radiátoru přišel jeho stav a nastaví ho do svojí vnitřní reprezentace řízení. Ta je uložena v instanci třídy *InternalState*, kterou lze vidět níže. Pokud tedy dojde ke změně stavu, odešle se pokyn k zapnutí nebo vypnutí boileru.

```
class InternalState:
  def __init__(self):
    self.rs = [0 for i in range(number_of_radiators)]
    self.prev = 0
    self.curr = 0

  def change(self, radiator_number, value):
    self.rs[radiator_number - 1] = 1 if '1' in value else 0
    self.prev = self.curr
    self.curr = 0 if len(filter(lambda i: i == 1, self.rs)) == 0 else 1

    return self.prev != self.curr
```

Tato třída uchovává stav radiátorů v poli *rs*. Radiátory jsou označeny R1 a R2, takže na indexu 0 bude stav radiátoru R1 a na indexu 1 stav radiátoru R2. Proměnné *prev* a *curr* uchovávají předchozí a aktuální stav výstupu pro boiler. Metoda *change* tedy nastaví hodnotu daného radiátoru a změní stav pro boiler na "1" pokud je alespoň jeden radiátor ve stavu "1" (zapnutý), pokud jsou oba dva ve stavu "0", bude výstup pro boiler "0".

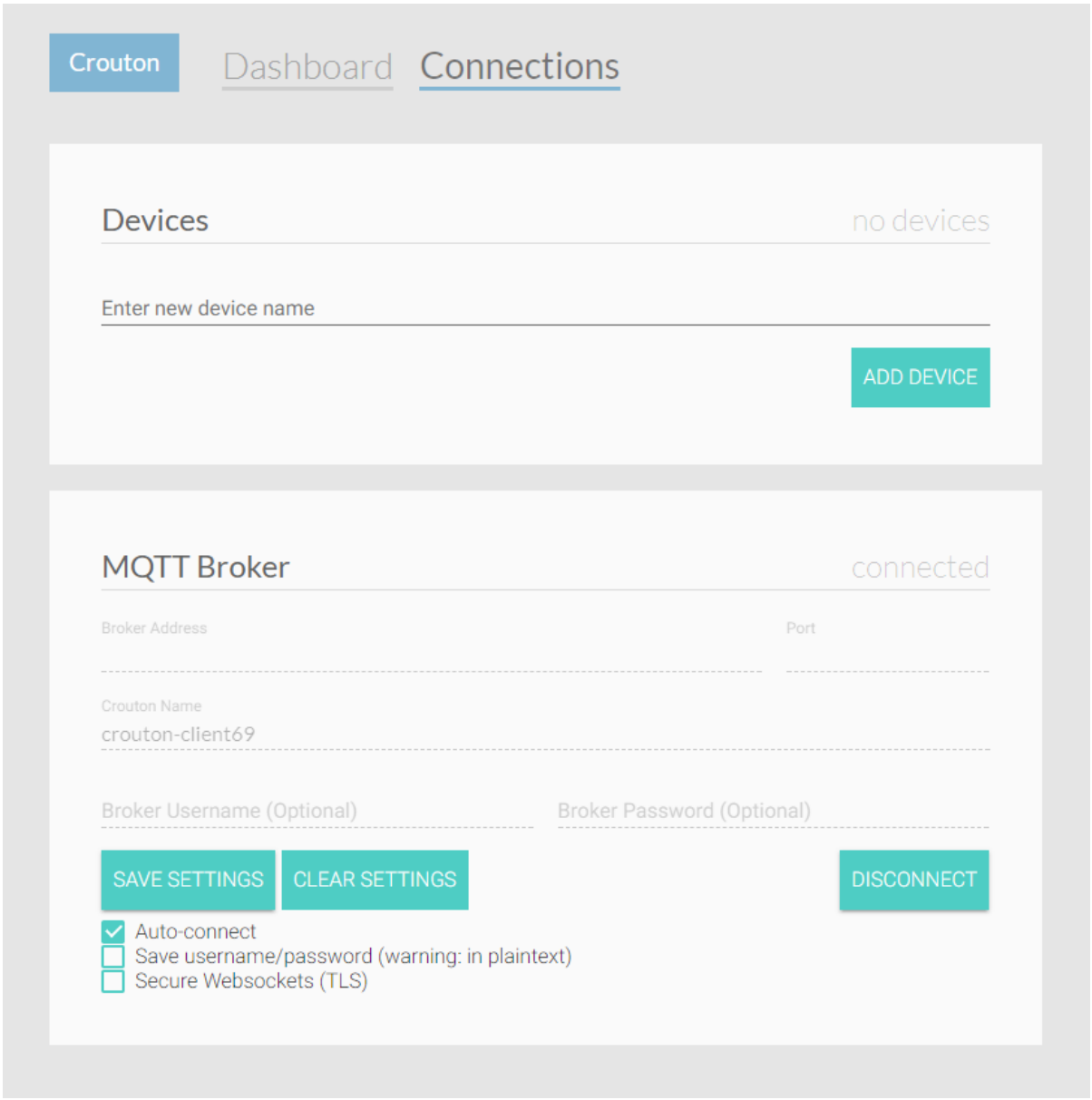
6.3 Dashboard

Crouton je projekt, který je založen na několika javascriptových technologiích. Jako buildovací nástroj používá *grunt*[1]. Crouton vyžadoval několik změn pro použití v demonstračním příkladu. Změny se týkaly požadavku, že je potřeba jednotlivé zobrazovací prvky rozdělit do skupin, aby nebyly všechny na jedné hromadě. Navíc to dává dobrý základ pro autentizaci a role, kdy určité pohledy mohou být zobrazeny jen určitým uživatelem. Bezpečnost a role ale nejsou v demonstračním příkladu řešeny. Technologie, které přímo souvisely s prováděnými změnami jsou:

- *pug*[2] - šablonovací engine umožňující psát HTML kód v úspornějším tvaru.
- *polymer*[3] - javascriptová knihovna, která pomáhá vytvářet HTML elementy.
- *packery*[4] - javascriptová knihovna pro tvorbu dynamických *layoutů*.
- *less*[5] - pre-procesor pro kaskádové styly, to znamená rozšířené možnosti zápisu CSS.

Crouton se samozřejmě připojuje k MQTT, a to pomocí websocket protokolu. Jakmile se připojí, je třeba přidat *device*, což je z pohledu Croutonu množina zobrazovaných komponent daná konfiguračním souborem, který byl probírán v kapitole 6.1.3. V souvislosti s PNVM a celkově demonstrační aplikací bude Crouton mít vždy jen jeden *device*, kde budou jednotlivé komponenty rozděleny do pohledů. Nastavení *device* s hodnotou *pnos-device* pro demonstrační příklad se provede v záložce *Connections* a bude přístupné po připojení se k MQTT brokeru. Ukázáno na následujícím obrázku.

[1] https://gruntjs.com/
[2] https://pugjs.org/api/getting-started.html
[3] https://www.polymer-project.org/
[4] https://packery.metafizzy.co/
[5] http://lesscss.org/



Obrázek 6.3: Nastavení MQTT v Croutonu

Další položka v menu je *Dashboard*, která zobrazuje komponenty. Pokud nebude Crouton připojený k MQTT, bude stránka prázdná. Až když získá konfiguraci od supervisora, stránka se naplní komponentami. Nemodifikovaný Crouton by zobrazil všechny komponenty najednou, ale bude přidána podpora pro pohledy. Základní věc spočívá v přidání druhého menu, z kterého se budou pohledy měnit. To se provede v souboru *crouton-dashboard.pug*, kde přidáme následující:

```
paper-menu(selected="{{viewSelected}}").device-menu
  template(is="dom-repeat", items="{{views}}")
    paper-item.item {{item.name}}
```

Elementy začínající na *paper* jsou elementy Polymeru, a klíčové slovo *template* v tomto případě definuje iteraci nad prvky v seznamu *views*. Prvek, který bude při načtení stránky vybraný je daný indexem *viewSelected*. Tyto tři řádky se v případě *views* definovaných podle demonstračního příkladu přeloží do HTML následovně:

```
<paper-menu ...>
  <div ...>
    <paper-item ...>test</paper-item>
    <paper-item ...>teplomery</paper-item>
    <paper-item ...>termostaty</paper-item>
    <paper-item ...>radiatory_boiler</paper-item>
    <paper-item ...>watchdog_status</paper-item>
  </div>
</paper-menu>
```

Atributy elementů jsou pro jednoduchost nahrazeny třemi tečkami. Element *paper-menu* a *paper-item* nejsou klasickými HTML elementy, ale s tím si poradí Polymer.

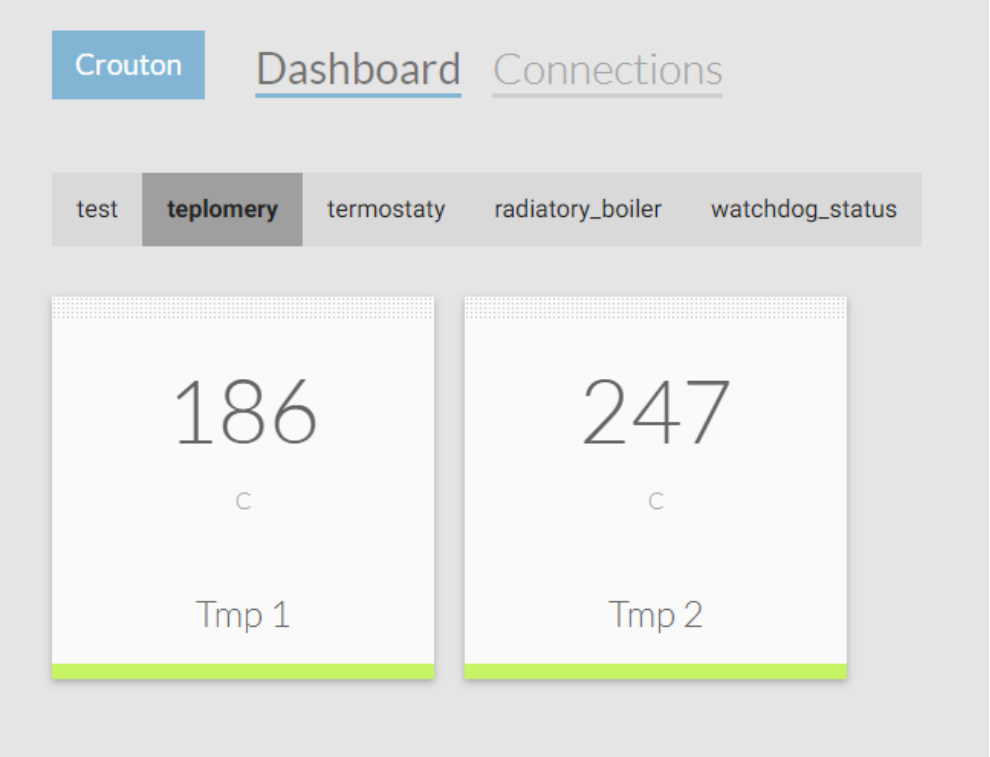
```
Polymer({
  ready: function() {
    ...
    this.views = [];
    this.onChange = function(next, prev){
      this.reset();
      for (var elementName in this.cardIndex) {
        if (this.cardIndex.hasOwnProperty(elementName)) {
          var element = this.cardIndex[elementName];
          var address = element.getAttribute("end-point-name");
          if (this.views[this.viewSelected].values.includes(address)) {
            this.push("cardsToBeAdded",{element: element, card:
                      element.nodeName.toLowerCase()});
          }
        }
      }
      if(this.currentPage == "dashboard"){
        this.async(this.addPackeryCards,100);
      }
    }
  },
  properties: {
    viewSelected: {
      type: Number,
      value: 0,
      observer: "onChange"
    },
    ...
  },
...
```

Inicializační funkce Polymeru definuje atributy a funkce, které budou přístupné i z definice HTML elementů i z ostatních funkcí. Zde jsou vidět úvodní hodnoty *views* a *viewSelected*, které používá *paper-menu*. Atribut *viewSelected* je inicializován na index 0 a je mu nastavena funkce *onChange*, která se bude volat vždy, když se hodnota atributu změní.

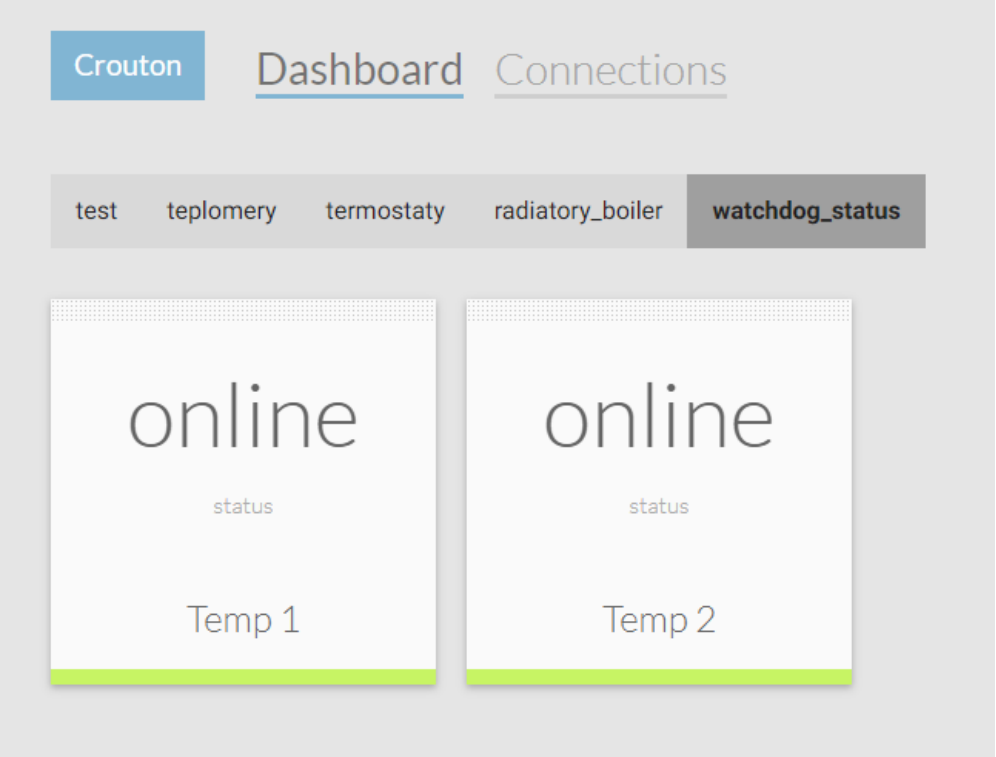
Funkce *onChange* prvně vymaže všechny zobrazené elementy a znovu inicializuje Packery kontejner na úvodní hodnotu. Poté projde všechny karty, které si Crouton drží z nastavení a podle vybraného *view* je přidá do Packery kontejneru. Nakonec dá pokyn Packery kontejneru, aby se překreslil.

```
this.splice("views", 0, this.views.length);
for (var view in deviceJson.views) {
  if (deviceJson.views.hasOwnProperty(view)) {
    this.push('views', { name: view, values: deviceJson.views[view] });
  }
}
```

Výše uvedený úryvek kódu je přidán do funkce, která je spouštěna při inicializaci Croutonu, když přijme svoji konfiguraci včetně definice pohledů. Tato úprava je nutná, aby se hned na začátku zobrazili pouze karty defaultně vybraného pohledu, což je pohled s indexem 0. Výsledek zobrazení sledovaných hodnot demonstrační aplikace je ukázán na obrázcích níže.



Obrázek 6.4: Crouton, pohled na teploměry



Obrázek 6.5: Crouton, pohled na stav tepoměrů od *watchdoga*

6.4 Běh aplikace

Jakmile existuje veškerá konfigurace, lze spustit všechny python moduly a PNVM. Jde o soubory *all_mqtt.py* a *run_pn_node* v tomto pořadí. V obou případech lze sledovat standardní výstup na zprávy, které se zde objevují.

Úryvek výstupu *run_pn_node*:

```
['DomoticzRead', 'Temp1']
['DomoticzRead', 'Temp2']
['DomoticzWrite', 'Boiler', '0']
['DomoticzWrite', 'R2', '0']
['DomoticzWrite', 'R1', '0']
['DomoticzRead', 'Temp1']
['DomoticzRead', 'Temp2']
['DomoticzWrite', 'Boiler', '0']
['DomoticzWrite', 'R2', '0']
['DomoticzWrite', 'R1', '0']
```

Výpis ukazuje aktivitu sítí pro teploměry, radiátory a boiler. Jde o volání externích skriptů, které buď zapisují nebo čtou. *DomoticzRead* je pro demonstrační příklad nastaven tak, aby generoval náhodná čísla v rozmezí 180-250, to znamená náhodnou teplotu v rozmezí 18 °C a 25 °C. *DomoticzWrite* pouze vrátí stejnou hodnotu, která měla být zapsána, jako potvrzení, že se podařilo hodnotu zapsat a aktuátor se přepnul. Jedná se o hodnoty "0" a "1", které znační vypnutí respektive zapnutí.

Úryvek z výstupu *all_mqtt.py*:

```
2017-05-20 18:55:14.775324 [cache] >>> writing: pnos/in/pnos-test/Thermost
2017-05-20 18:55:15.049760 [cache] >>> writing: pnos/in/pnos-test/R2/SET {
2017-05-20 18:55:15.827693 [cache] >>> writing: pnos/out/pnos-test/R2/STAT
2017-05-20 18:55:15.828068 [router] >>> pnos/out/pnos-test/R2/STATUS -> /o
2017-05-20 18:55:15.985946 [cache] >>> writing: pnos/out/pnos-test/R1/STAT
2017-05-20 18:55:15.986266 [router] >>> pnos/out/pnos-test/R1/STATUS -> /o
2017-05-20 18:55:16.632310 [cache] >>> writing: pnos/out/pnos-test/Temp1/S
2017-05-20 18:55:16.632685 [router] >>> pnos/out/pnos-test/Temp1/STATUS ->
2017-05-20 18:55:18.440727 [supervisor] >>> device pnos-test disconnected
```

Nejaktivnější je typicky router a cache, protože oba reagují na data ze senzorů, která přicházejí často. Na konci je vidět zpráva od supervisor, kterou poslal, když se ukončil běžící skript *run_pn_node*.

7 Závěr

7.1 Semestrální projekt

První fází této práce bylo prozkoumání a nastudování problematiky IoT a to hlavně v rámci existujícího komplexního softwaru v podobě Domoticz a OpenHAB. Bylo potřeba vše zprovoznit, vyzkoušet a pochopit principy daných řešení. To probíhalo v prostředí Raspberry Pi, což potenciálně bude primární volba pro uzly chytrého domu. Součástí této fáze také bylo zaměření se na MQTT protokol a možnosti jeho použití.

Další fáze se věnovala PNVM. Referenční Petriho sítě jsou velmi silným nástrojem pro modelování systémů. Jakmile je definovaná referenční síť, PNVM ji dokáže interpretovat bez potřeby další implementace. Existuje zde také velký potenciál z hlediska IoT v dynamické rekonfiguraci, kde je obecně výhodné mít možnost lehce změnit chování systému.

Následovala fáze hledání existujících softwarových řešení pro integraci PNVM a IoT. Open-sourcových řešení existuje velké množství, ale najít vhodné není složité. Všechna diskutovaná řešení jsou dobře dokumentovaná a v případě potřeby dobře rozšiřitelná - to platí především pro dashboard část řešení, neboť zde se v budoucnu bude řešit propojení s databází.

Vznikla první myšlenka architektury systému chytrého domu založeného na PNVM uzlech s komunikací pomocí MQTT protokolu. Tato myšlenka se bude dále rozšiřovat hlavně v oblasti integrace databáze, ale také v oblasti přístupu systému z hlediska uživatelů. O tomto nebyla v návrhu explicitní zmínka, ale může se stát, že k jednomu systému budou přistupovat různí uživatelé, kteří by si neměli navzájem vidět svá data. Řešení tohoto problému bude součástí výsledné diplomové práce.

Diplomová práce bude dále pokračovat v rozšiřování základního návrhu se souběžným testováním navržených rozšíření a výsledkem bude implementace systému pro chytrý dům s uzly na bázi PNVM, komunikačním protokolem MQTT a využití existujících open-sourcových řešení. Součástí výsledku bude i demonstrační aplikace používající výsledný systém.

7.2 Diplomová práce

Z vize ze semestrálního projektu se do diplomové práce nakonec nedostalo řešení přístupu různých uživatelů, pouze se diskutovaly základní možnosti. K čemu ale došlo, bylo rozšiřování návrhu a extenzivní testování prvků společně s PNVM.

Stěžejní části bylo dotáhnout tu část komunikace, která probíhá mezi PNVM a MQTT. Z toho důvodu se objevila i menší adaptace v PNVM samotném, a poté se lehce implementovala integrace převodu zpráv generových PNVM do tvaru vhodném pro MQTT a zpět. Docela dobře se podařilo zakomponovat adresování z PNVM zpráv do MQTT *topiců*, což ulehčilo práci především při implementaci python modulů.

Vylepšení se dočkal i python wrapper pro PNVM, který je nyní lépe odladěn a především je plně integrovaný s MQTT.

Docela velká část práce spočívala ve zlepšení běhu PNVM bez nutnosti externě posílat konfigurační zprávy ručně přes MQTT a přesunutí úvodní konfigurace při spuštění PNVM. Jedná se samozřejmě o python moduly, které tuto roli zastávají. Výhoda je, že tyto moduly můžou běžet typicky na stejném serveru jako broker, kde budou mít také konfigurační soubory mimo jiné pro

každé PNVM připojující se k tomu serveru. Tím pádem je konfigurace centralizovaná a PNVM instance mohou být distribuované.

Dále vznikl Crouton dashboard pro zobrazení dat ze senzorů a ovládání aktuátorů chytrého domu. Crouton používá celkem různorodé javascriptové frameworky, což byla zajímavá zkušenost modifikovat kód pro účely této práce. Dalším krokem, který by mohl významně rozšířit tuto práci z hlediska dashboardu, by byla integrace možnosti vývoje Petriho sítí přímo z dashboardu. V textu byl zmíněn *translator*, který překládá kód Petriho sítí namodelovaných v nástroji Renew do PNBC, ten je esenciální pro tuto možnost.

S dashboardem také souvisí bezpečnost, možnost přístupu různých uživatelů, což by mohlo být další významné implementační rozšíření této práce. V této práci byly diskutovány role, jaké by se mohly objevit. Tato problematika rozhodně stojí za pozornost a bude nutností, pokud by se řízení chytrého domu Petriho sítěmi mělo rozšířit mezi běžné uživatele.

Posledním větším tématem, které se příliš neposunulo v rámci této práce je integrace databáze. Původní myšlenka byla taková, že se do databáze bude ukládat veškerá MQTT komunikace. Tato myšlenka vykrystalizovala do podoby, že v databázi se budou ukládat data s časovým razítkem zejména pro tvorbu grafů z hodnot ze senzorů. Tato práce připravila ukládání dat do databáze, už je zbývá nějak využít.

Literatura

[1]. *Domoticz* [online]. 2016 [cit. 2016-12-26]. Dostupné z: https://domoticz.com/

[2]. *OpenHAB* [online]. 2016 [cit. 2016-12-26]. Dostupné z: http://www.openhab.org/

[3]. *MQTT Protokol* [online]. 2016 [cit. 2016-12-26]. Dostupné z: http://mqtt.org/

[4]. *Mosquitto dokumentace* [online]. [cit. 2016-12-26]. Dostupné z: https://mosquitto.org/documentation/

[5]. *SCADA systémy* [online]. [cit. 2016-12-26]. Dostupné z: https://inductiveautomation.com/what-is-scada

[6]. RICHTA Tomáš a JANOUŠEK Vladimír. *Operating System for Petri Nets-Specified Reconfigurable Embedded Systems.* In Computer Aided Systems Theory - EUROCAST 2013. Berlin Heidelberg: Springer Verlag, 2013, s. 444-451. ISBN 978-3-642-53855-1.

[7]. MINÁŘ Michal. *Interpret Petriho sítí pro řídicí systémy s procesorem Atmel*, diplomová práce, Brno, FIT VUT v Brně, 2013

[8]. *Mosquitto broker* [online]. 2016 [cit. 2017-01-02]. Dostupné z: https://mosquitto.org/

[9]. *SQLite* [online]. 2016 [cit. 2017-01-06]. Dostupné z: https://sqlite.org/

[10]. *PostgreSQL* [online]. 2016 [cit. 2017-01-06]. Dostupné z: https://www.postgresql.org/

[11]. *CouchDB* [online]. 2016 [cit. 2017-01-06]. Dostupné z: http://couchdb.apache.org/

[12]. *RRDTool* [online]. 2016 [cit. 2017-01-06]. Dostupné z: http://oss.oetiker.ch/rrdtool/

[13]. *InfluxDB* [online]. 2016 [cit. 2017-01-06]. Dostupné z: https://docs.influxdata.com/influxdb/v1.1/

[14]. *Home.pi* [online]. 2016 [cit. 2017-01-08]. Dostupné z: https://github.com/denschu/home.pi

[15]. *Crouton* [online]. 2016 [cit. 2017-01-08]. Dostupné z: https://github.com/edfungus/Crouton