



**FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE**

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název: Inteligentní zabezpečovací systém garáže: Nadřazený systém
Student: Bc. Ondřej Červenka
Vedoucí: Ing. Martin Daňhel
Studijní program: Informatika
Studijní obor: Návrh a programování vestavných systémů
Katedra: Katedra číslicového návrhu
Platnost zadání: Do konce letního semestru 2018/19

Pokyny pro vypracování

Dle analýzy vyberte vhodnou platformu (Raspberry Pi, ZYNQ apod.) a navrhnete a implementujete systém (ve formě aplikace/bitstreamu) pro správu jednotlivých garáží.

Navržený systém na základě příchozích událostí vyhodnotí situaci a zareaguje vhodnou akcí (např. jedna z garáží se nebude hlásit, v jedné z garáží čidla zahlásí, že hoří, případně neautorizovaný vstup apod.).

Systém bude umožňovat správu podřízených systémů jednotlivých garáží (ty nejsou součástí práce). Pro komunikaci (eth/wifi) mezi jednotlivými garážemi a navrhovaným systémem navrhnete rozhraní na základě rešeršní analýzy.

Navrhnete webové rozhraní pro správu garáží, které bude umožňovat zobrazit současný stav jednotlivých garáží (otevřeno, zamčeno, obsazeno, volno), pokusů o útok včetně historie událostí. Do webového rozhraní je umožněn pouze autorizovaný přístup.

Systém navrhnete s ohledem na bezpečnost, spolehlivost a rozšiřitelnost.

Celé zařízení otestujete.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Hana Kubátová, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 31. ledna 2018

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
KATEDRA ČÍSLICOVÉHO NÁVRHU



Diplomová práce

Nadřazený systém pro správu garáže

Bc. Ondřej Červenka

Vedoucí práce: Ing. Martin Daňhel

17. března 2018

Poděkování

Děkuji panu Ing. Martinu Daňhelovi za čas, který mi věnoval a zejména za cenné rady a odborné vedení mé diplomové práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 17. března 2018

.....

České vysoké učení technické v Praze

Fakulta informačních technologií

© 2018 Ondřej Červenka. Všechna práva vyhrazena.

Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Červenka, Ondřej. *Nadřazený systém pro správu garáže*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato diplomová práce se zabývá tvorbou zabezpečovacího systému pro správu garáží. Konkrétně jde o návrh a implementaci nadřazeného systému, který sbírá a zpracovává data zaslaná podřízenými systémy v jednotlivých garážích.

Výsledná aplikace komunikuje s podřízenými systémy pomocí HTTP protokolu a umožňuje správu garáží přes webové rozhraní. Celý nadřazený systém je možné provozovat na Raspberry Pi.

Klíčová slova bezpečnost, webová aplikace, Raspberry Pi

Abstract

Sem doplňte ekvivalent abstraktu Vaší práce v angličtině.

Keywords security, web application, Raspberry Pi

Obsah

Úvod	1
1 Teoretický základ	3
1.1 Základní pojmy	3
1.2 Bezpečnost	4
1.3 Spolehlivost	4
2 Analýza	5
2.1 Struktura systému	6
2.2 Výběr komunikačního protokolu	8
2.3 Ukládání dat	14
2.4 Zasílání notifikací	15
2.5 Programovací jazyk pro tvorbu systému	15
2.6 Výběr platformy	16
3 Návrh	23
3.1 Návrhový vzor MVC	23
3.2 <i>Model</i>	24
3.3 <i>View</i>	29
3.4 <i>Controller</i>	31
3.5 Autentizace uživatele	35
4 Implementace	37
4.1 Struktura aplikace	37
4.2 Implementace <i>modelu</i>	37
4.3 Implementace <i>view</i>	39
4.4 Implementace <i>controlleru</i>	40
4.5 Autentizace uživatele	40
5 Testování	41

Závěr	43
Literatura	45
A Nasazení	49
B Uživatelská příručka	51
B.1 První přihlášení	51
B.2 Hlavní stránka	52
B.3 Stránka garáže	54
B.4 Uživatelské nastavení	55
C Seznam použitých zkratk	57
D Obsah přiloženého CD	59

Seznam obrázků

2.1	Základní struktura systému	7
2.2	Příklad struktury protokolu MQTT	11
2.3	Raspberry Pi 3	17
2.4	Přípravek Zybo Zynq-7000	18
2.5	Blokové schéma využití IP jádra Xilibus	19
2.6	Struktura systému provozovaného na virtuálním serveru	22
3.1	Struktura MVC aplikace	24
3.2	Diagram vyhodnocování stavu garáže	28
3.3	Vrchní úroveň případů užití webového rozhraní	29
3.4	Případy užití správy garáží	30
3.5	Případy užití zobrazení událostí	31
3.6	Případy užití uživatelského nastavení	32
B.1	Přihlašovací formulář webového rozhraní	51
B.2	Výzva ke změně hesla	52
B.3	Hlavní stránka aplikace	52
B.4	Stránka garáže	54

Seznam tabulek

2.1	Srovnání platforem Raspberry Pi 3 a Zybo Zynq-7000	20
-----	--	----

Úvod

Tato práce se zabývá zabezpečením garážového komplexu. V něm je potřeba jednotlivé garáže zajistit jednak proti nedovolenému vniknutí, jednak proti nebezpečí požáru. Je tedy nutné sledovat události indikující vznik těchto hrozeb (například detekce kouře či pohybu). K tomu slouží zařízení umístěná v garážích, která sledují stav svého okolí pomocí čidel a jsou schopna detekovat nebezpečí.

Tato zařízení nepracují samostatně, ale odesílají data o zaznamenaných událostech centrální jednotce – nadřazenému systému¹. Tento nadřazený systém zpracovává příchozí informace a varuje uživatele o potenciálních hrozbách.

Kromě toho také vytváří historii zachycených událostí. Ta může mít informativní hodnotu pro provozovatele komplexu (například údaje o vytíženosti jednotlivých garáží), nebo také posloužit při policejním vyšetřování (časy otevření dveří či zacyhčení pohybu).

Cílem práce je vytvořit a otestovat tento nadřazený systém. Výsledná aplikace bude komunikovat pomocí WiFi či Ethernetu s podřízenými systémy (monitorovacími zařízeními v garážích). Na základě získaných dat pak bude udržován stav jednotlivých garáží a vytvářena historie událostí.

Systém bude poskytovat webového rozhraní pro administraci. V tom bude možné přidávat a odebírat podřízené systémy, zobrazovat jejich stav a zaznamenané události.

Vzhledem k povaze zadání je nutné systém navrhnout s ohledem na zabezpečení přenášených informací před odposloucháváním či manipulací. Též je nutné autorizovat uživatele přistupující do webového rozhraní.

Dalším důležitým požadavkem je snadná rozšiřitelnost o nové funkce. Systém by mělo být možné v budoucnu doplnit o možnost správy rozdílných

¹V textu práce budu dále používat označení „nadřazený systém“ pro tuto centrální jednotku a „podřízený systém“ pro monitorovací zařízení, která jsou umístěna v jednotlivých garážích.

podřízených systémů (například subsystémy pro sledování skladových zásob) či integraci s mobilní aplikací. Bude tedy potřeba navrhnout vhodné komunikační rozhraní pro předávání informací mezi systémem a jeho klienty.

V práci se chci zaměřit na tvorbu aplikace na jedné konkrétní hardwarové platformě, jako je například jednodeskový počítač Raspberry Pi. Aplikace spolu s touto platformou by pak měla tvořit kompletní zařízení, které bude možné po základní konfiguraci (připojení do WiFi sítě, nastavení hesla) hned nasadit.

Výsledné řešení by však mělo být dostatečně nezávislé na zvolené platformě. Tudíž by neměl být problém spustit systém například na osobním počítači či virtuálním serveru.

V analytické části (2) práce tedy přiblížím proces výběru vhodné platformy, komunikačního protokolu a dalších prvků systému. Také stručně popíšu podřízený systém (garážové čidlo), se kterým budu dále pracovat. Další části mapují návrh (3) systému na základě této analýzy, jeho implementaci (4), nasazení (A) na zvoleném hardwaru a testování (5).

Teoretický základ

1.1 Základní pojmy

do ty teorie dat spis veci jako protokoly, koncepty, nebo obecny technologie k softwaru dat citaci primo v textu

- uživatel – use case diagramy, nebo ty možná až v návrhu
- klic
- api – to už ale je ve zkratkách, jestli to tady budu nak rozepisovat tak popsat taky json
- client/server
- tcp/ip
- osi
- http/https, požadavek, metody, návratové kody, hlavičky požadavku atd...
- hash, u hashe zmínit bcrypt a eventuelně jiné bezpečné algoritmy, sul
- mqtt
- certifikát, k tomu veřejný a soukromý klíč, self-signed jen třeba lehce zmínit, protože je už dost popsanej v té http sekci
- vícevláknová obsluha
- nadřazený systém
- podržovaný systém
- mitm attack

1. TEORETICKÝ ZÁKLAD

- csrf
- publisher/subscriber
- webové rozhraní
- unix time
- cloud
- attack surface
- relační databáze, databázový schema, vztahy (1:n, atd)

1.2 Bezpečnost

1.3 Spolehlivost

Analýza

Před samotným návrhem a implementací nadřazeného systému je potřeba přesněji definovat požadavky na jeho vlastnosti a chování. Z hlediska chování by systém měl poskytovat následující funkce:

- Systém má být schopen komunikovat s podřízenými systémy pomocí WiFi či Ethernetu (v rámci místní sítě). Komunikace je založena na zasílání událostí zaznamenaných podřízenými systémy.
- Systém má uchovávat zaznamenané události, včetně data a času přijetí a podřízeného systému, který událost vytvořil.
- Systém má umožňovat správu pomocí webového rozhraní, které by mělo umožňovat následující funkce:
 - Přidat (registrovat) či odebrat podřízený systém.
 - Zobrazit stav registrovaných podřízených systémů.
 - Zobrazit zaznamenané události.
 - Měnit nastavení jednotlivých podřízených systémů.
 - Měnit uživatelské nastavení.
- Systém má být schopen informovat provozovatele při poplachu či poruše.

Dále jsou od nadřazeného systému požadovány tyto vlastnosti:

- **Bezpečnost** – jak komunikace s podřízenými systémy, tak přístup do uživatelského rozhraní by měl probíhat přes zabezpečený komunikační kanál.
- **Spolehlivost** – způsob komunikace i systém jako celek by měl být dostatečně spolehlivý.

- **Presistence** – data v systému by měla být uchovávána persistentním způsobem.
- **Nezávislost** – systém by neměl být příliš závislý na externích službách a alespoň v omezené míře fungovat i bez přístupu k internetu.

Na základě těchto požadavků je tedy potřeba zvolit vhodné nástroje pro implementaci systému. To je v první řadě komunikační protokol použitý pro komunikaci s podřízenými systémy. Tento protokol by měl být dostatečně robustní, rozšířený a podporovat šifrování. Také by s jeho pomocí mělo být možné bezpečně ověřit identitu podřízených systémů. Výběrem protokolu se zabývá sekce 2.2.

Další důležitá volba je způsob uchování zachycených událostí a dalších údajů. Data je nutné uchovávat persistentně, nejlépe v obecně používaném formátu, který umožní další zpracování i mimo nadřazený systém. Výběr úložiště je popsán v sekci 2.3.

Nadřazený systém má být také schopen upozornit uživatele v případě hrozícího nebezpečí nebo výpadku podřízeného systému. To je možné provést mnoha způsoby, jejichž možnosti jsou shrnuty v sekci 2.4.

Nakonec zbývá zvolit programovací jazyk a případně frameworky pro tvorbu aplikace. Vhodných jazyků je dnes velké množství, jde tedy spíše o otázku osobní preference. Volbou programovacího jazyka se zabývá sekce 2.5.

Z těchto požadavků vyplývá software potřebný k implementaci nadřazeného systému, jehož dostupnost je hlavní požadavek při volbě hardwarové platformy v sekci 2.6.

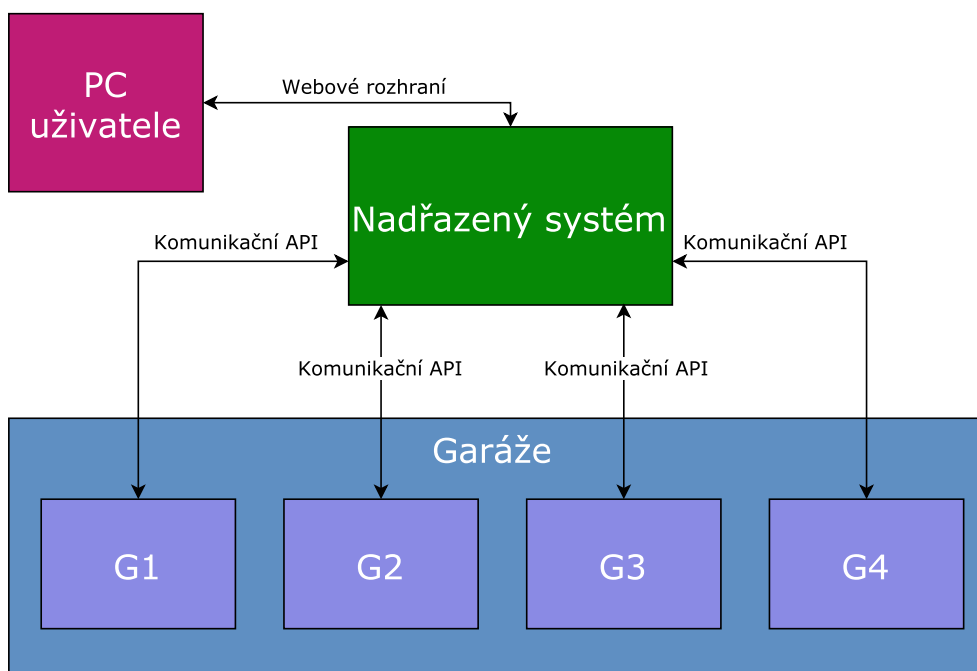
2.1 Struktura systému

Struktura celého systému je naznačena na obrázku 2.1. [Podřízené systémy](#) komunikují s [nadřazeným](#) na základě událostí. Nadřazený systém tyto události zpracovává a upravuje podle nich stav garáží v evidenci.

Zaznamenané události jsou také uchovávány v historii událostí, spolu s dalšími metadaty jako čas přijetí nebo podřízeného systému, který událost vytvořil.

Komunikace mezi podřízeným a nadřazeným systémem je postavena na modelu *client/server*. Nadřazený systém provozuje server zvoleného protokolu (viz sekci 2.2), ke kterému se podřízené systémy připojují. Komunikaci tedy vždy iniciuje podřízený systém. S možností zasílání nevyžádaných zpráv podřízeným systémům v této práci nepočítám, mohl by to však být námět pro další rozšíření.

Další, kdo přistupuje do systému, je [uživatel](#). Přes webové rozhraní může sledovat stav garáží a historii událostí. Také zde může spravovat klíče, které slouží pro přístup ke komunikačnímu API systému. Přístup do webového rozhraní je zabezpečen heslem.



Obrázek 2.1: Základní struktura systému. Podřízené systémy využívají komunikační API k zasílání událostí. Nadřazený systém tyto události zaznamenává a upravuje podle nich stav garáží. Uživatel nadřazený systém ovládá pomocí webového rozhraní

2.1.1 Podřízený systém

Podřízený systém je zařízení umístěné v každé garáži, které sleduje stav okolí pomocí těchto senzorových vstupů:

- detekce kouře,
- detekce pohybu,
- stav dveří.

Základní požadavek na podřízený systém je schopnost komunikace přes Ethernet či WiFi pomocí protokolu zvoleného v sekci 2.2. Jinak může být hardware prakticky libovolný.

Návrh a implementace podřízeného systému není součástí této práce. Jeho vývojem se zabývá Miroslav Váňa ve své bakalářské práci na Katedře číslicového návrhu Fakulty informačních technologií ČVUT.

2.1.2 Událost

Událost slouží jako základní komunikační jednotka mezi nadřazeným a podřízeným systémem. V případě překročení mezních hodnot senzorů se jednotlivé podřízené systémy okamžitě hlásí nadřazenému systému. Kromě toho také v pravidelných intervalech odesílají kontrolní hlášení.

Vyhodnocení události je provedeno nadřazeným systémem. Podřízený systém tedy hlásí každou událost (například otevření dveří), aniž by nějak zkoumal její závažnost.

2.2 Výběr komunikačního protokolu

Nejdříve je nutné určit způsob komunikace, který bude systém používat. Díky tomu se budu při vybírání platformy moci ujistit, že jsou dostupné vhodné knihovny a další software.

Nadřazený systém bude se svými klienty (monitorovací zařízení v jednotlivých garážích) komunikovat přes WiFi nebo Ethernet. Základem komunikace bude TCP/IP protokol, je však potřeba zvolit vhodný protokol z aplikační vrstvy OSI modelu, který na něm bude stavět.

2.2.1 Vlastní protokol

Jedna z možností je implementovat vlastní protokol pomocí TCP/IP socketů. Toto řešení se mi však nezdá příliš vhodné, neboť nepřináší žádné významné výhody, naopak se s ním pojí řada komplikací.

Pro vlastní protokol by bylo nutné vytvořit robustní server, který zvládá obsluhu více klientů najednou. Dále by vzhledem k citlivosti přenášených dat bylo nutné implementovat nějakou formu šifrování. Tyto velmi obsáhlé problémy přitom řeší většina dnešních protokolů.

Další nevýhodou je nutnost implementace klientské části protokolu při vytváření nových zařízení spravovaných nadřazeným systémem. To do jisté míry omezuje jeho rozšiřitelnost.

2.2.2 Protokol HTTPS

Další možnost je využít ke komunikaci protokol HTTPS. V tomto případě by klienti komunikovali se systémem pomocí HTTP metod jako například `get` nebo `post`.

Jelikož součástí požadavků na systém je i webové uživatelské rozhraní, bude v každém případě nutné použít webový server pro jeho provoz. Ten by pak bylo možné využít i k poskytnutí API pro komunikaci systému s podřízenými systémy.

Vhodný webový server (jako například Apache [1]) zajistí vícevláknovou obsluhu všech klientů. Protokol se také postará o šifrování přenášených dat, je však nutné získat certifikát k ověření pravosti serveru (viz sekci 2.2.2.1).

Certifikát bude potřeba zajistit i v případě, že komunikace s klienty nebude postavena na tomto protokolu. Je totiž nutné také zabezpečit webové rozhraní, například kvůli ověření totožnosti uživatele při přihlašování. Nutnost pořízení certifikátu tedy nepředstavuje nevýhodu oproti jiným protokolům.

API realizované pomocí tohoto protokolu je poměrně snadno rozšiřitelné. Pro nově implementovanou operaci stačí definovat URL a případně formát přenášených dat.

Výhodou je také snadná implementace na straně klienta, tedy podřízeného systému. Knihovny realizující klientskou část protokolu jsou dostupné na většině populárních platform jako například Arduino (s Ethernet *shieldem*, oficiální knihovna EthernetClient [2]) nebo ESP8266 (knihovna esp8266wifi [3]).

2.2.2.1 Certifikáty pro provoz HTTPS

Pro provoz HTTPS serveru lze použít například certifikáty certifikační autority Let's Encrypt, které jsou poskytovány zdarma [4]. Kromě toho dodává Let's Encrypt také automatizačního klienta Certbot [5] pro snadné nasazení a aktualizaci jejich certifikátů. Bohužel certifikáty jsou vydávány pouze na doménu [4], což komplikuje použití v místní síti.

Jiná možnost je použití *self-signed* certifikátu [6]. Tento certifikát není podepsaný žádnou certifikační autoritou, ale pouze vlastníkem certifikátu. Může tedy sloužit k šifrování komunikace (poskytuje veřejný klíč), ale je zranitelný vůči *man-in-the-middle* útoku [6].

Self-signed certifikát však lze použít k šifrování komunikace na uzavřené lokální síti, za předpokladu, že je server s certifikátem (přesněji s jeho soukromým klíčem) dostatečně zabezpečen [6].

Nevýhodou tohoto řešení je nedůvěra webových klientů (certifikát není podepsán certifikační autoritou a nelze tedy ověřit jeho pravost), což by ovlivnilo přístup k uživatelskému rozhraní a API systému. V případě webového rozhraní by prohlížeč zobrazil varování o neznámém certifikátu. To by však mohl uživatel ignorovat.

Podřízené systémy by při zasílání požadavků museli přeskočit krok ověření totožnosti serveru. Jak toho dosáhnout například v knihovně Requests (umožňující vytváření HTTP požadavků [7]) pro jazyk Python [8] je naznačeno v ukázce 1.

Tento jazyk a knihovna byly zvoleny z důvodů kompaktnosti ukázkového kódu, v jiných jazycích lze tuto verifikaci obejít obdobným způsobem.

```
>>> import requests
>>> r = requests.get('https://test.local/hello', verify=False)
>>> r.status_code
200
```

Ukázka 1: Vytvoření HTTPS požadavku v knihovně Requests, bez ověření totožnosti serveru

2.2.2.2 Autentizace klientů na HTTPS

Přístup k API nadřazeného systému by měl být povolen pouze ověřeným klientům. Díky tomu bude možné zabránit například zasílání nepravdivých informací z neznámých zdrojů.

Jednoduchou autentizaci přes HTTPS lze realizovat například pomocí generování API klíčů. Pro každý podřízený systém bude vygenerován klíč, kterým se při zasílání požadavku systém prokáže. Seznam platných klíčů by byl udržován v databázi nadřazeného systému. Klíče by uživatel mohl přidávat nebo odebírat (například v případě odcizení podřízeného systému) pomocí webového rozhraní.

Tyto klíče by také bylo nutné nahrát a uchovávat na podřízených systémech. Detaily tohoto procesu by záležely na platformě těchto systémů. Například u Arduina by šlo klíč nahrát z uživatelova počítače pomocí sériové linky (s USB převodníkem) a udržovat ho v paměti EEPROM, která je uchována i po odpojení napájení [9].

Také by bylo možné implementovat v nadřazeném systému „registrační mód“, který by bylo možné dočasně povolit ve webovém rozhraní. V tomto módu by systém po přijetí speciálního API požadavku vygeneroval nový klíč. Ten by si uložil do své databáze platných klíčů, a také ho v odpovědi zaslal žádajícímu zařízení. Pokud by mód povolen nebyl, odpověděl by systém chybovým kódem, například 403 – *Forbidden*. Zaslání požadavku z podřízeného systému by mohlo být provedeno stisknutím tlačítka.

Tento přístup by byl pravděpodobně uživatelsky příjemnější, přináší však potencionální bezpečnostní rizika. Například pokud by uživatel zapomněl mód vypnout, systém by byl otevřený k registraci nežádoucích zařízení. Takový problém by se však dal řešit například automatickou deaktivací módu po uplynutí časového limitu.

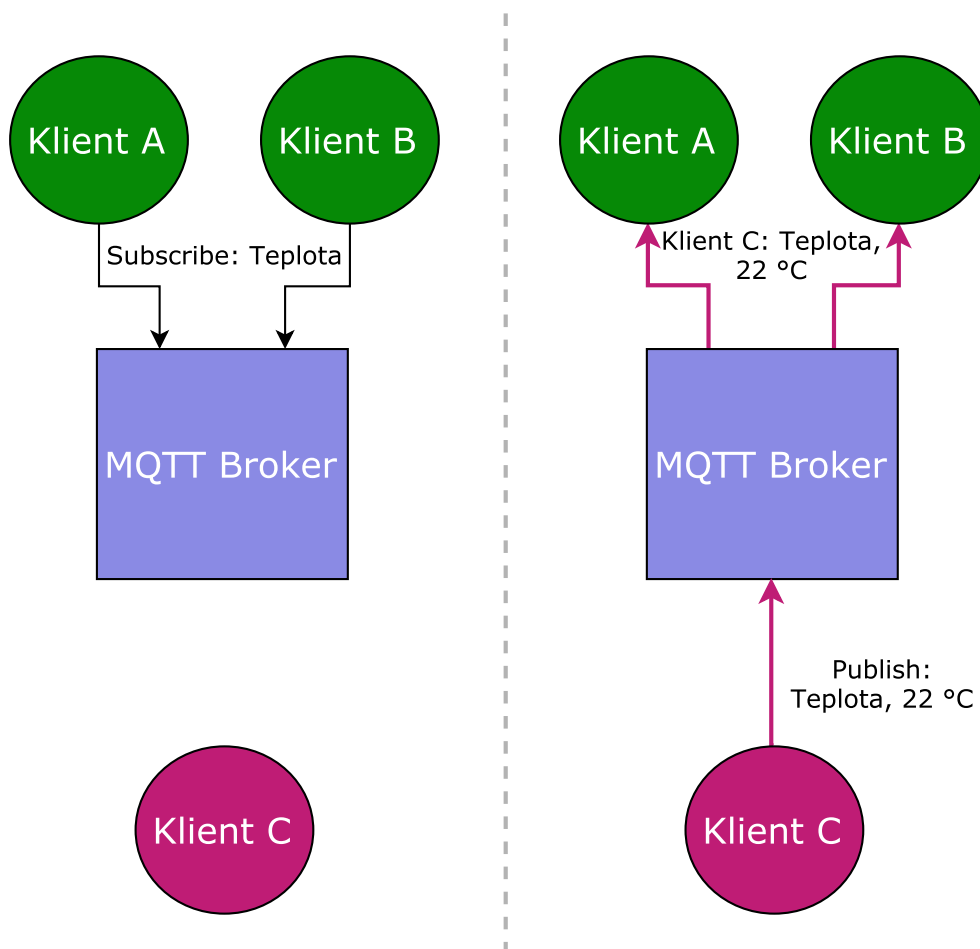
Útočník snažící se získat klíč k API by také mohl periodicky zkoušet registrační požadavek a čekat na aktivaci módu. Obrana proti tomuto útoku by byla složitější, šlo by například filtrovat IP adresy s příliš častými požadavky.

Obecně vycházím z toho, že i v případě registrace nežádoucího zařízení nemůže toto zařízení krátkodobě způsobit výraznější škody – do databáze nadřazeného systému může pouze zasílat nová data, která jsou navíc vázána k jeho identitě (API klíči). Nemůže tedy získávat data od jiných podřízených systému či měnit jejich záznamy. Neautorizované zařízení se také objeví v seznamu registrovaných API klíčů, kde může být snadno odhaleno.

2.2.3 Protokol MQTT

MQTT je komunikační protokol založený na modelu *publisher/subscriber*, určený pro použití v prostředí s omezenými zdroji (malý výkon procesoru, omezená paměť atd.) [10].

Komunikace mezi jednotlivými klienty v systému je zprostředkována pomocí centrály, nazývané *broker*. Ta spravuje adresy – *topics* – na kterých mohou klienti publikovat či odebírat zprávy.



Obrázek 2.2: Příklad struktury protokolu MQTT [11]. Klienti A a B jsou *subscriber* *topicu* „Teplota“. Když klient C na tuto adresu publikuje novou zprávu, *broker* se postará o její doručení všem *subscriberům*

Na obrázku 2.2 tedy klienti A a B začnou odebírat *topic* „Teplota“. Když pak klient C publikuje zprávu na tuto adresu, *broker* se postará o doručení všem odebírajícím klientům.

2. ANALÝZA

Adresy je možné hierarchicky strukturovat následujícím způsobem [11]:

- Lze tvořit skupiny, například:
 - `/senzory/obyvak/teplota`.
 - `/senzory/kuchyne/vlhkost`.
- Zpráva je nutné publikovat na jednoznačnou adresu.
- Při odebírání je možné použít modifikátory pro specifikování skupiny adres:
 - Modifikátor `+` – odpovídá libovolnému jednomu stupni hierarchie. Například pro odebírání všech senzorů vlhkosti lze použít adresu `/senzory/+/vlhkost`.
 - Modifikátor `#` – odpovídá libovolnému počtu libovolných stupňů. Adresa `/senzory/#` tedy slouží k odebírání všech dat.

V případě této práce by tedy jak nadřazený systém, tak podřízené systémy byly klienty *brokeru*. Podřízené systémy by publikovaly naměřená data, která by nadřazený systém odebíral. Tím by vznikla obdoba *client/server* modelu, zmíněného v sekci 2.1.

Samotný *broker* by pak mohl běžet souběžně s nadřazeným systémem na zvolené platformě (například open-source *broker* Mosquitto je dostupný na řadě platform, včetně Raspberry Pi [12]).

Protokol podporuje tři možnosti QoS [10]:

- Nejvýše jedno doručení – tento mód pouze odešle zprávu, není zahrnut žádný opakovací mechanismus pro případ nedoručení.
- Alespoň jedno doručení – v tomto módu je zaručeno doručení zprávy, ta však může být doručena vícekrát.
- Přesně jedno doručení – zde je ošetřeno i duplicitní doručování zpráv.

Použití sofistikovanějších metod doručení má vliv na výkon, a proto se v některých případech vyplatí zvolit nižší úroveň QoS (například při posílání idempotentních zpráv). Pro tuto práci bych však pravděpodobně zvolil záruku přesně jednoho doručení.

2.2.3.1 Šifrování a autentizace na MQTT

V této části se budu zabývat prostředky pro šifrování komunikace, které jsou dostupné v *brokeru* Mosquitto.

První možnost je pro zabezpečení komunikace využít certifikáty, podobně jako u HTTPS. Zde by se pravděpodobně také využil *self-signed* certifikát

(blíže popsany v sekci 2.2.2.1). Mosquitto navíc také vyžaduje kořenový certifikát certifikační autority [13]. Při použití *self-signed* certifikátů by bylo nutné tuto autoritu vytvořit a používané certifikáty u ní podepsat (pro bližší informace viz [14]). Kořenový certifikát by také bylo nutné distribuovat klientům.

Kromě certifikátů lze pro šifrování použít i PSK. V tom případě *broker* a jeho klienti používají pro zašifrování komunikace společný klíč (známý jak klientovi, tak *brokeru*). Různí klienti přitom mohou mít různé klíče. [15]

Bohužel podpora PSK v MQTT klientech není příliš rozšířená. PSK je možné použít v knihovně libmosquitto, určené pro C/C++ (s vazbami pro Python). U této knihovny se mi však podařilo najít pouze manuálovou stránku (viz [16]), bez informací o jejím dalším vývoji či udržování. Modul poskytující vazby do Pythonu byl nicméně předán projektu Paho [17].

Paho poskytuje implementace MQTT klientů pro mnoho platform (včetně například Arduina [18]). Dokumentace klientů pro C++ a Python však možnost šifrování pomocí PSK vůbec nezmiňuje [19] [20].

Tyto možnosti lze použít i k autentizaci klientů *brokeru*. Při použití certifikátů lze v konfiguračním souboru Mosquitta zvolit `require_certificate` [15]. Poté bude od klienta vyžadován certifikát prokazující jeho totožnost. Při použití PSK lze k autentizaci využít sdílený klíč (*broker* odmítne klienty s neplatnými klíči) [15]. Kromě toho je možno použít také autentizaci pomocí uživatelského jména a hesla, která je součástí MQTT protokolu, případně klienty neověřovat vůbec (a pouze šifrovat komunikaci) [15].

2.2.4 Závěr výběru protokolu

V sekcích 2.2.2 a 2.2.3 jsem se blíže podíval na dva poměrně rozšířené protokoly aplikační vrstvy, které by bylo možné použít pro tvorbu nadřazeného systému.

Pokud by mezi požadavky na systém bylo zahrnuto zasílání nevyžádaných zpráv podřízeným systémem (jak je zmíněno v sekci 2.1), zvolil bych pravděpodobně protokol MQTT. V tom je tato funkcionality velmi snadno implementovatelná – stačí aby podřízené systémy odebíraly *topic*, na kterém by nadřazený systém publikoval zprávy.

Jelikož se však v této práci zabývám systémem, který zprávy pouze přijímá a zaznamenává, rozhodl jsem se pro HTTPS. Nasazení tohoto protokolu je o něco snazší (není potřeba na zařízení instalovat *broker* a zařizovat certifikační autoritu – stačí *self-signed* certifikát) a s jeho použitím mám více zkušeností. Také se částečně uvolní požadavky na volbu platformy (webový server bude potřeba v každém případě, při volbě HTTPS jako komunikačního protokolu mezi systémy tedy nebude nutný žádný další software).

Každopádně bude mým cílem navrhnout výslednou aplikaci tak, aby rozhraní pro podřízené systémy realizované pomocí HTTPS bylo možné snadno nahradit MQTT rozhraním.

K zabezpečení komunikace (včetně webového rozhraní) použiju *self-signed* certifikát. Hlavní důvod je požadavek na použití v místní síti, bez zaručeného

přístupu k internetu. Toto rozhodnutí nemá vliv na návrh a implementaci systému, pouze na jeho nasazení – konkrétně konfiguraci webového serveru.

Pokud by provozovatel plánoval mít systém přístupný z internetu (přes registrovanou doménu), musí v každém případě k zabezpečení použít certifikát podepsaný důvěryhodnou certifikační autoritou. Pak stačí pouze v konfiguračním souboru webového serveru nahradit *self-signed* certifikát podepsaným certifikátem. Není tedy nutné provádět změny v kódu aplikace.

2.3 Ukládání dat

Zaznamenané události bude potřeba persistentně uchovávat. Zde by šly využít jednoduché textové logy, vhodnější však bude zvolit nějaký databázový systém – například kvůli širším možnostem zpracování naměřených údajů.

Z dostupných možností mě zaujal SQLite, což není klasický databázový stroj s modelem *client/server*, ale místo toho tvoří součást programu, který databázi používá. Přístup k datům je realizován pomocí přímého čtení/zápisu do databázového souboru na disku. Díky tomu má malé nároky na diskový prostor a operační paměť. [21]

Jelikož bude nadřazený systém pravděpodobně provozován na hardwaru s omezenými zdroji, představují tyto vlastnosti nezanedbatelnou výhodu. Použití SQLite také zjednoduší nasazení aplikace, neboť nebude nutné vytvářet a konfigurovat databázový server.

2.3.1 Velikost databáze

Vzhledem k požadavku na spolehlivost systému je vhodné odhadnout předpokládaný růst velikosti databázového souboru. Příliš velká databáze by mohla vyčerpávat paměť zařízení, a způsobit tak jeho pád. Také by mohla negativně ovlivnit dobu odezvy nadřazeného systému.

Pokud by se ukázalo, že při dlouhodobém provozu začíná být databáze neúnosně velká, bylo by nutné omezit množství uchovávaných dat, například pomocí front zaznamenaných událostí.

Při odhadu jsem vycházel z předpokladu, že největší objem dat v databázi budou představovat zaznamenaná kontrolní hlášení podřízených systému. Pokud se budou podřízené systémy hlásit každou hodinu, představuje 24 záznamů denně na každou garáž, což by mělo být řádově více než u jiných událostí. Například u pravděpodobně druhé nejčastější události, tedy otevření/zavření dveří, předpokládám nejvýše jednotky denně.

Událost kontrolního hlášení by měla obsahovat tyto údaje:

- Identifikátor události v databázi.
- Identifikátor garáže, ke které událost patří.
- Datum a čas zaznamenání události.

- Datum a čas dalšího plánovaného hlášení.

Jako identifikátory pravděpodobně budou stačit 32bitová celá čísla. Datum je možné v SQLite ukládat buď jako textový řetězec ve formátu YYYY-MM-DD HH:MM:SS.SSS, 64bitové reálné číslo, či 32bitové celé číslo – *Unix Time* [22]. Z toho vyplývá, že jedno kontrolní hlášení bude mít velikost alespoň $4 * 32 = 128$ bitů.

Pokud nadřazený systém monitoruje 30 garáží, kde každá odesílá kontrolní hlášení každou hodinu, jsou každý den vytvořeny záznamy o velikosti $30 * 24 * 96 = 98304$ bitů, tedy asi 98 Kb. Dá se tedy předpokládat, že velikost databázového souboru neporoste nijak dramaticky.

Tento výpočet představuje pouze odhad předpokládané velikosti. Skutečná velikost bude záležet na zvolené implementaci, neměla by se však výrazně lišit od tohoto odhadu.

2.4 Zasílání notifikací

Nadřazený systém má být schopen zasílat uživatelům notifikace o stavu jednotlivých garáží. Notifikace mohou informovat o důležité události (například detekce kouře) nebo o výpadku podřízeného systému (v případě promeškání plánovaného kontrolního hlášení).

Notifikace jsou zasílány pomocí e-mailu na seznam adres zvolených uživatelem. K tomu je zapotřebí vytvořit účet, který by mohl nadřazený systém použít k odesílání.

Zde se nabízí možnost provozovat e-mailový server s účtem přímo na zařízení, na kterém poběží nadřazený systém. Provoz takového serveru je však velmi náročný, a to nejen kvůli složité počáteční konfiguraci, ale také z hlediska dlouhodobé údržby [23]. Tento přístup by tak kladl neúměrné nároky na uživatele systému.

Alternativa je použít nějakého poskytovatele e-mailových služeb, jako například Gmail od společnosti Google. Zde by si uživatel vytvořil účet určený k odesílání notifikací, a poté autorizoval nadřazený systém k odesílání zpráv z tohoto účtu.

Tímto je částečně porušen požadavek na nezávislost (viz sekci 2). Při zasílání notifikací je však nutný přístup k internetu i v případě použití vlastního e-mailového serveru a nezávislost na externí službě nevyváží komplikace spojené s jeho provozem.

2.5 Programovací jazyk pro tvorbu systému

Pro tvorbu systému jsem se rozhodl použít programovací jazyk Python [8]. S tímto jazykem mám nejvíce zkušeností co se týče implementace webových

aplikací. Je také dostatečně rozšířený, takže výsledný systém bude možné nasa-
dit na poměrně širokém spektru platform bez nutnosti složitějšího portování.

K vytvoření webového rozhraní i API pro podřízené systémy jsem zvolil
framework Flask [24]. Hlavní důvod jsou opět předchozí zkušenosti s tímto
frameworkem. Flask také dává více volnosti při návrhu aplikace než například
také velmi rozšířený framework Django [25].

2.6 Výběr platformy

Pro realizaci systému je nutné zvolit vhodnou platformu. Jelikož je cílem práce
vytvořit fyzické zařízení, rozhodl jsem se jako základ použít některý z jedno-
deskových počítačů, které jsou v dnešní době na trhu. Tyto počítače bývají
cenově velmi dostupné a zároveň poskytují dostatečný výkon a podporu pro
provoz systému.

Při výběru počítače byla nejdůležitějším kritériem podpora softwaru po-
třebného k implementaci monitorovacího systému. Na základě předchozí ana-
lýzy je tedy vyžadován následující software:

- Webový server Apache2.
- Databázový systém SQLite.
- Programovací jazyk Python 3.
 - Webový framework Flask.

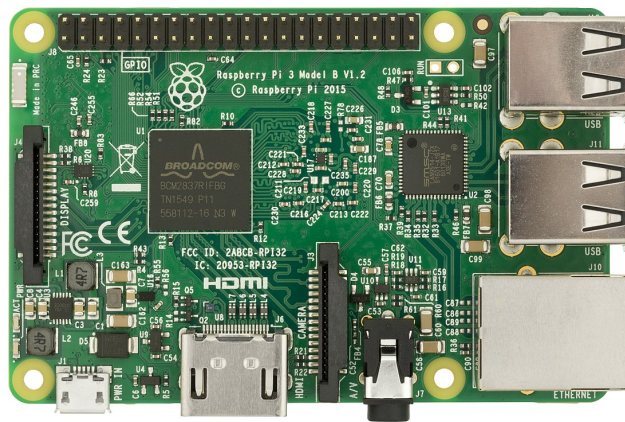
Pro provoz tohoto softwaru bude potřeba plnohodnotný operační systém,
což vylučuje platformy využívající jednoduché mikrokontroléry, jako například
Arduino Uno. Kromě toho je nutné připojení k síti pomocí Ethernetu nebo
WiFi.

V dalších sekcích jsem se blíže podíval na jednodeskové počítače Raspberry
Pi (sekce 2.6.1) a Zybo Zynq-7000 (sekce 2.6.2) a zvážil jejich výhody a nevý-
hody pro implementaci systému.

2.6.1 Raspberry Pi

Raspberry Pi je velmi rozšířený jednodeskový počítač. Jeho poslední verze,
Raspberry Pi 3, je postavena na SoC Broadcom BCM2837 s čtyřjádrovým
procesorem ARM Cortex A53, který je až o 50 % rychlejší než procesor před-
chozí verze [26].

Dále nová verze přináší vlastní WiFi modul [26], není tedy nutné se spo-
léhat na externí moduly. Kromě toho je možné počítač připojit k síti pomocí
Ethernetového portu. Ten je omezený na 100 Mb/s [26], to by však vzhledem
k objemu dat přenášných mezi nadřazeným a podřízenými systémy nemělo
představovat problém.



Obrázek 2.3: Raspberry Pi 3 (obrázek převzat z https://en.wikipedia.org/wiki/Raspberry_Pi)

Deska také obsahuje čtyři USB a jeden HDMI port [26]. Ty nejsou pro implementovaný systém zásadní, nicméně při počáteční konfiguraci zařízení (například nastavení WiFi hesla) může být připojení monitoru a klávesnice pro některé uživatele pohodlnější než použití SSH či sériové linky. Připojený monitor se také hodí při řešení problémů se startem operačního systému.

Raspberry Pi 3 bohužel nemá vlastní bateriově zálohovaný RTC obvod a k udržování času využívá protokol NTP [27]. K tomu je však zapotřebí internetové připojení. Jelikož by zařízení mělo být možné používat i v síti bez přístupu k internetu, je nutné připojit externí RTC obvod, například pomocí I2C sběrnice [27]. Poté je možné systémové hodiny synchronizovat bez internetového připojení pomocí tohoto obvodu.

S počítačem je možné použít množství operačních systémů, z nichž nejrozšířenější je pravděpodobně Raspbian, linuxový systém postavený na Debianu [28]. Pro ten jsou dostupné všechny potřebné softwarové balíčky popsané v sekci 2.6. Jelikož počítač nemá žádné vlastní úložiště, je nutné operační systém provozovat na vložené SD kartě [26].

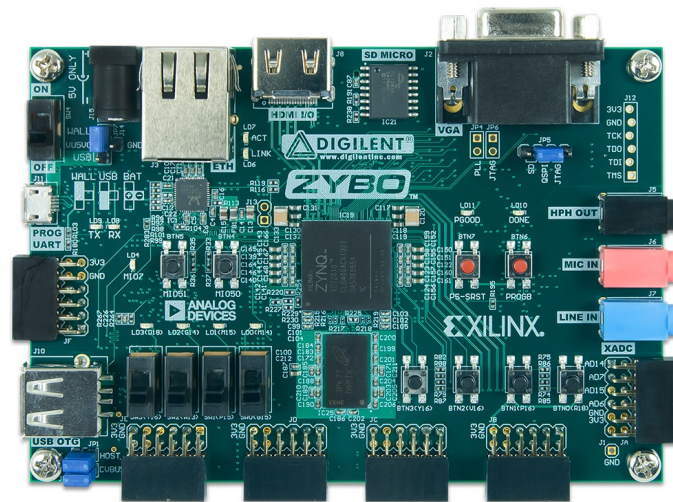
Jednou z výhod tohoto počítače je obrovské množství podporovaných hardwarových periférií a knihoven pro ně. V této práci pravděpodobně využiju pouze zmíněný RTC obvod, případné další rozšiřování systému (například o vestavěný LCD displej) bude na Raspberry Pi pravděpodobně snadnější než na jiných platformách.

Kromě široké podpory je hlavní výhodou Raspberry Pi jeho cena. Poslední verze se pohybuje kolem 1200 Kč. K celkovým nákladům na systém je ještě třeba připočítat cenu RTC obvodu a SD karty. Zde počítám s použitím již připraveného modulu s obvodem PCF8523 (pro bližší informace o modulu viz [27]). Ten vyjde asi na 200 Kč. Jako úložiště by měla plně dostačovat 16GB

2. ANALÝZA

SD karta, která se dá pořídit za 200 Kč. Celkové náklady na hardware systému se tedy měly pohybovat kolem 1600 Kč².

2.6.2 Zybo Zynq-7000



Obrázek 2.4: Přípravek Zybo Zynq-7000 (obrázek převzat z <https://www.xilinx.com/products/boards-and-kits/1-4azfte.html>)

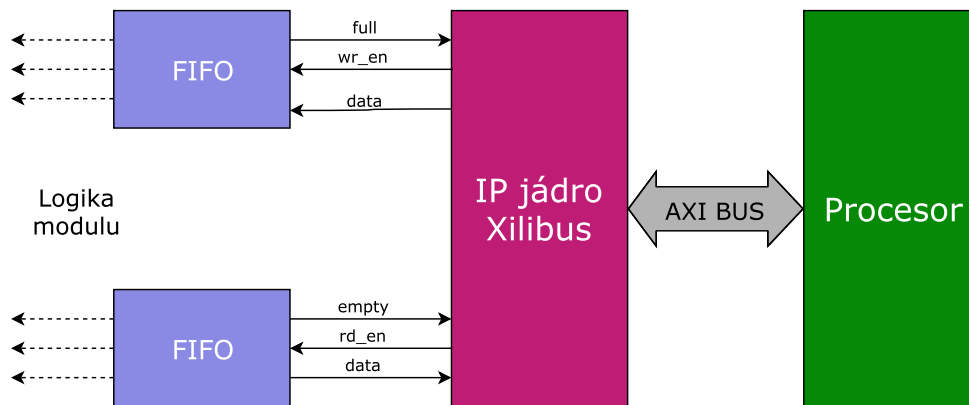
Tento přípravek od společnosti Digilent je postavený na SoC Xilinx Zynq Z-7010. Hlavní předností tohoto čipu je kombinace dvoujádrového procesoru ARM Cortex A9 s FPGA odpovídající sérii Artix-7 [29].

Díky tomu je možná těsná integraci mezi aplikací běžící na procesoru a výkonnými, úzce specializovanými moduly, které jsou syntetizované na FPGA. Tento přístup, kdy je hardware a software systému vyvíjen souběžně se označuje jako *hardware/software codesign* [30].

Pro SoC ze série Zynq vznikla linuxová distribuce Xilinx, vycházející z Ubuntu. Kromě plnohodnotného operačního systému (včetně například grafického rozhraní) poskytuje Xilinx také ovladače pro komunikaci s FPGA pomocí AXI sběrnice [31]. K tomu využívá **IP jádro Xilibus**, které funguje jako adaptér mezi **procesorem** a FPGA modulem (viz obrázek 2.5). Ten pak ke komunikaci může využívat standardní **FIFO** fronty a nemusí se zabývat AXI sběrnici [31].

Jelikož Xilinx staví na Ubuntu (konkrétně na verzi 12.04 LTS [31]), neměl by být problém nainstalovat software potřebný pro provoz nadřazeného systému (viz sekci 2.6).

²Ceny převzaté z obchodů Alza (<https://alza.cz>) a SnailShop(www.snailshop.cz), v únoru 2018.



Obrázek 2.5: Blokové schéma využití IP jádra Xilibus [31]. Procesor přípravku komunikuje pomocí AXI sběrnice s jádrem Xilibus. To požadovaná data zapíše či přečte z FIFO fronty příslušného modulu

Přípravek je možné připojit k síti pomocí Ethernetového portu, který podporuje rychlost až 1Gb/s [29]. WiFi připojení by bylo možné realizovat pomocí USB modulu. Dále přípravek obsahuje HDMI a VGA port, audio konektory, čtyři tlačítka, čtyři přepínače a slot pro SD kartu [29].

Na přípravku je také k dispozici 128 MB flash paměti [29], k provozu tedy teoreticky není potřeba SD karta. Tato paměť by však pravděpodobně nestačila k instalaci vhodného operačního systému a potřebného softwaru. I zde by tedy bylo nutné použít SD kartu.

Stejně jako Raspberry Pi tato deska postrádá RTC obvod. Firma Digilent však dodává externí obvod, který lze připojit pomocí Pmod rozhraní [32].

Nevýhodou této desky (především v porovnání Raspberry Pi) je její cena. Ta se pohybuje okolo 4000 Kč³. K tomu je nutné přičíst náklady na SD kartu a RTC obvod, případně i WiFi modul. Cena celého zařízení by se tedy pohybovala v rozmezí 4500 až 5000 Kč.

2.6.3 Závěr výběru platformy

Jak vyplívá z tabulky 2.1, Raspberry Pi 3 poskytuje znatelně výkonnější procesor a více paměti za méně než třetinu ceny desky Zybo. Hlavní přidaná hodnota této platformy tedy spočívá v integraci s FPGA, ta má však v případě této práce pouze velmi omezené využití.

FPGA by bylo možné využít například k šifrování úložiště, vzhledem k předpokládaným objemům dat by však zrychlení oproti softwarovému šifrování neospravedlnilo vysokou cenu přípravku. Na druhou stranu vyšší výkon pro-

³Cena podle produktové stránky <https://www.xilinx.com/products/boards-and-kits/1-4azfte.html>, v únoru 2018.

	Raspberry Pi 3	Zybo Zynq-7000
SoC	Broadcom BCM2837	Xilinx Zynq Z-7010
Procesor	ARM Cortex A53 4 jádra, 1,2 GHz	ARM Cortex A9 2 jádra, 650 MHz
RAM	1024 MB LPDDR2	512 MB DDR3
FPGA	–	ekvivalent řady Artix-7
Úložiště	SD karta	128 MB Flash, SD karta
Síť	100 Mb/s Ethernet, WiFi	až 1 Gb/s Ethernet
Cena	1200 Kč	4000 Kč

Tabulka 2.1: Srovnání platforem Raspberry Pi 3 a Zybo Zynq-7000

[26] [29]

cesoru u Raspberry Pi může mít pro nadřazený systém význam, například z hlediska odezvy uživatelského rozhraní.

Jako platformu pro implementaci nadřazeného systému jsem tedy zvolil Raspberry Pi 3, především kvůli příznivé ceně, výrazně lepšímu poměru cena/výkon (pro tuto práci), a také kvůli podpoře a rozšiřitelnosti.

2.6.4 Provoz aplikace na cloudové platformě

V této části bych chtěl popsat alternativu k provozu systému na dedikovaném zařízení, a to možnost využít virtuální server na některé *cloudové* platformě. Primární cíl práce je sice vytvořit nadřazený systém jako jednoúčelové zařízení (postavené na Raspberry Pi), nicméně provoz výsledné aplikace v *cloudu* může být v určitých situacích vhodnější řešení.

Jedním z možných uplatnění této varianty je monitorování více garážových komplexů. Místo lokálního nadřazeného systému by mohly všechny *podřízené systémy* z každého komplexu komunikovat s jedním *globálním systémem*, provozovaným na virtuálním serveru a dostupným z internetu, jak je naznačeno na obrázku 2.6.4. Při tomto provozu je však nutnost mít registrovanou doménu a zabezpečit spojení podepsaným certifikátem.

Virtuální servery nabízí například firma DigitalOcean. Cena serveru závisí na počtu výpočetních jader, dostupně RAM a velikosti úložiště. Nejlevnější konfigurace přijde na 5 dolarů měsíčně a nabízí jednojádrový procesor, 1 GB RAM a 25 GB SSD [33]. Předpokládám, že tento výkon by stačil pro základní provoz systému, v případě vyššího počtu podřízených systémů je však možnost zvyšovat dostupnou RAM a jádra CPU.

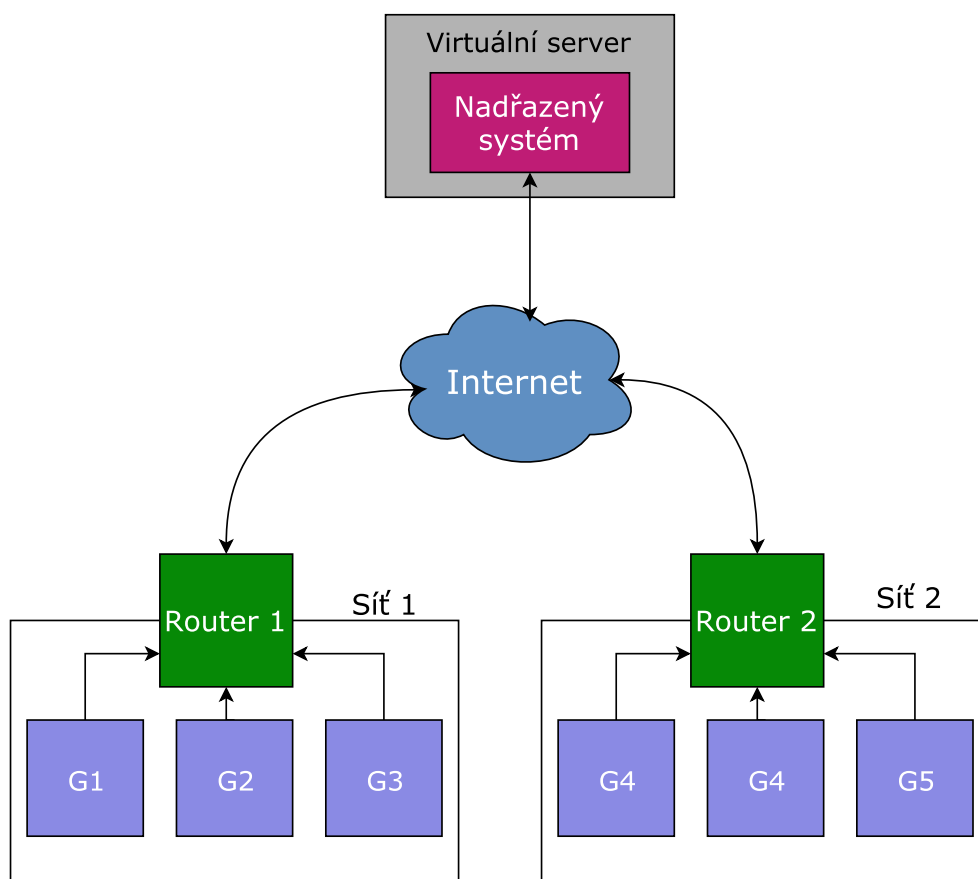
S těmito virtuálními servery lze použít řadu běžných linuxových operačních systémů jako Ubuntu, Debian či Fedora [34], dostupnost softwaru potřebného pro spuštění aplikace tedy není problém.

Hlavní nevýhodou tohoto přístupu je poněkud složitější nasazení a spuštění systému. Registrace domény, získání certifikátu a základní konfigurace webo-

vého serveru se dá sice částečně automatizovat (viz zmiňovaný Certbot [5]), pro většinu uživatelů bude však pravděpodobně snazší použít již připravené Raspberry Pi.

Také je složitější distribuovat adresu serveru s nadřazeným systémem podřízeným systémům. V místní síti může podřízený systém sám nalézt nadřazený systém testováním odpovědi zařízení v síti na registrační požadavek. Pokud by však byl nadřazený systém dostupný pouze na internetové doméně, bylo by nutné ji ručně zadat každému podřízenému systému.

Další problém může představovat hlavní výhoda tohoto řešení, a to přístupnost serveru z internetu. Ta významně zvyšuje *attack surface* celého nadřazeného systému, zvláště oproti alternativě využívající k propojení systémů pouze Ethernet u kterého má (na rozdíl od WiFi) provozovatel fyzický přehled o připojených zařízeních.



Obrázek 2.6: Struktura systému provozovaného na virtuálním serveru. Podřízené systémy nezasílají požadavky lokálnímu nadřazenému systému v místní síti, ale globálnímu nadřazenému systému, který je veřejně přístupný na internetu

Návrh

Tato kapitola se zabývá návrhem nadřazeného systému na základě předchozí analýzy z kapitoly 2. Nadřazený systém je navržen podle vzoru MVC, jehož jednotlivé části jsou popsány v sekcích 3.2, 3.3 a 3.4.

Poslední sekce 3.5 se zabývá návrhem autentizace uživatele při přihlašování do webového rozhraní nadřazeného systému.

3.1 Návrhový vzor MVC

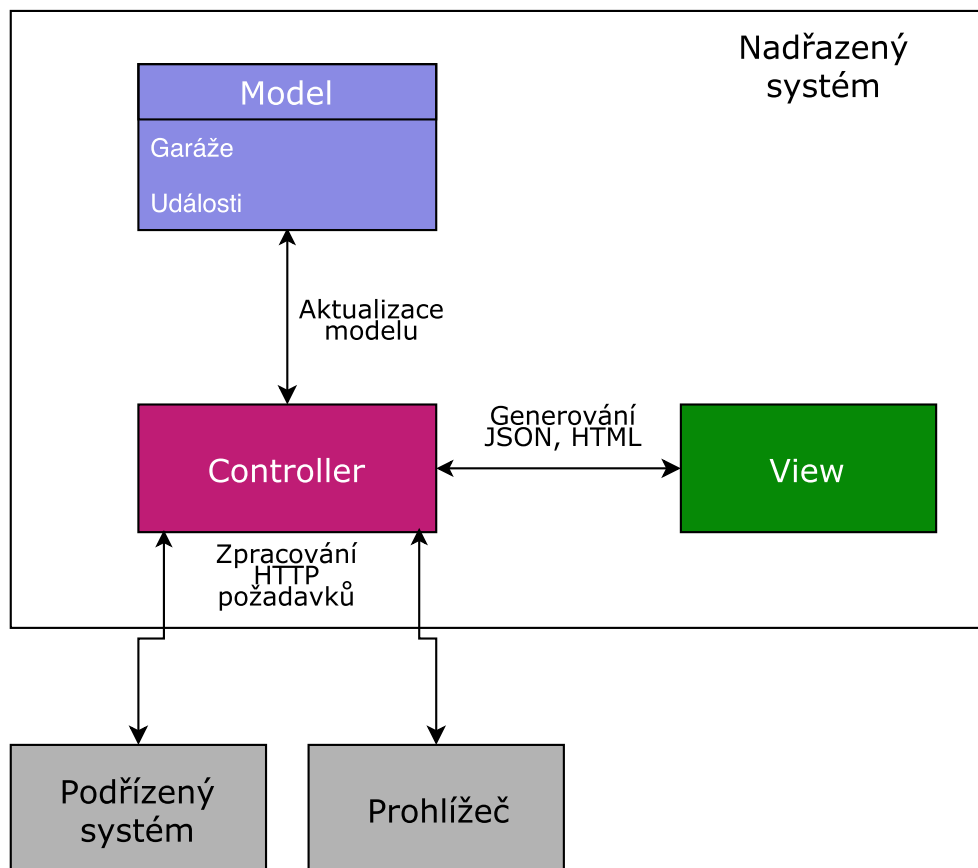
Struktura nadřazeného systému je vhodná k použití návrhového vzoru MVC, tedy *model-view-controller*. *Model* zde představují garáže (podřízené systémy), k nim vázané události a logika jejich vyhodnocování.

View je zobrazení těchto dat, tedy především generované HTML stránky webového rozhraní. Jako další *view* je možné považovat získávání dat (například ve formátu JSON) pomocí API nadřazeného systému, třeba při zasílání registračních klíčů podřízeným systémům.

Controller je pak část aplikace, která se stará o zpracování HTTP požadavků. Ty mohou přicházet jednak z uživatele prohlížeče, jednak od podřízených systémů. Na základě těchto požadavků pak *controller* posílá příslušné příkazy *modelu*. Struktura aplikace při použití vzoru MVC je naznačena na obrázku 3.1.

Hlavní motivací pro použití tohoto vzoru je snadná rozšiřitelnost. Pokud by například bylo potřeba aplikaci doplnit o komunikaci s podřízenými systémy pomocí MQTT, stačí pouze vytvořit vhodný *controller*. Ten pak může využívat *model* aplikace stejným způsobem jako HTTP *controller*.

Tento návrhový vzor popisuje pouze nadřazený systém spravující garáže. Kromě toho je součástí aplikace ještě autentizace uživatele při přístupu do uživatelského rozhraní. Tuto funkci jsem se rozhodl pojmout jako samostatnou komponentu, popsanou v sekci 3.5.



Obrázek 3.1: Struktura MVC aplikace. Uživatel i podřízený systém používají aplikaci pomocí *controlleru*. Ten definuje příslušná URL a zpracovává požadavky na ně – aktualizuje *model*, případně doručí odpovídající *view*

3.2 *Model*

Model představuje jádro nadřazeného systému. Je zde implementována vnitřní reprezentace uchovávaných dat, což jsou především zaznamenané události. Každá událost má také svého původce, tedy garáž (přesněji podřízený systém v této garáži).

Kromě toho *model* implementuje *business logiku* systému, jako je například vytváření nových garáží (registraci podřízených systémů) či reakce na příchozích událostech.

Ostatní části aplikace (*view* a *controller*) používají *model*, bez znalosti jeho vnitřní struktury, pomocí následujících operací:

- **Vytvoření garáže** – registrace nového podřízeného systému. V případě vytváření pomocí API (tj. přímo podřízeným systémem) vyžaduje ope-

race zapnutý registrační mód. Pokud je nová garáž vytvářena ve webovém rozhraní, zapnutý registrační mód není vyžadován.

- **Editace garáže** – například změna označení.
- **Smazání garáže** – smazáním garáže dojde k odstranění zaznamenaných událostí a zneplatnění příslušného API klíče.
- **Zneplatnění API klíče** – odepření přístupu podřízenému systému bez nutnosti smazání garáže a s ní spojených událostí.
- **Zapínání registračního módu** – v tomto módu je možné vytvářet nové garáže na základě požadavku od podřízeného systému. Bližší informace jsou v sekci 3.2.1.1. Registrační mód se po uplynutí časového limitu sám vypne.
- **Přístup k uloženým datům** – obecně operace typu získání všech garáží nebo všech událostí vázané ke konkrétní garáži.
- **Vytvoření události** – základní požadavek využívaný podřízenými systémy.

Vnitřně pak model na základě těchto operací autentizuje pomocí API klíčů požadavky podřízených systému, validuje zasláná data, spravuje databázi nadřazeného systému nebo obstarává zaslání notifikačních e-mailů.

3.2.1 Garáž – Garage

Třída **Garage** reprezentuje konkrétní podřízený systém a uchovává s ním spojená data:

- **id** – identifikace entity v databázi.
- **tag** – uživatelem zvolené označení garáže. To slouží pro snadnější orientaci ve webovém rozhraní (uživatel nemusí garáže rozlišovat podle nic neříkajícího id).
- **note** – poznámka pro další popis garáže.
- **api_key** – klíč umožňující přístup k API systému. Ten je také zaslán zařízení při jeho registraci. Generování a správa klíčů je blíže popsána v sekci 3.2.1.2.
- **last_report** – datum a čas posledního kontrolního hlášení.
- **next_report** – datum a čas dalšího očekávaného hlášení.
- **period** – perioda kontrolních hlášení.

- **doors** – stav dveří garáže (otevřeno/zavřeno).
- **state** – celkový stav garáže (v pořádku, nehlásí se atd.).
- Seznam událostí spojených s touto garáží.

3.2.1.1 Vytváření nových garáží

Nové garáže mohou v systému vznikat dvěma způsoby. První možnost je vytvoření nové garáže přímo v uživatelském rozhraní. Po vytvoření se zde zobrazí vygenerovaný API klíč, který je potřeba nahrát na příslušný podřízený systém. Způsob nahrávání by závisel na příslušném hardwaru (například sériová linka). Tato možnost je určena především pro podřízené systémy, které by nepodporovaly zaslání registračního požadavku a nevyžaduje zapnutý registrační mód.

Druhá možnost je použít registrační mód. V tom případě je nová garáž vytvořena na základě registračního požadavku podřízeného systému (pro bližší informace o tomto požadavku viz sekci 3.4.1). Vygenerovaný API klíč je při tom zaslán jako odpověď na požadavek, a není tedy nutné ho ručně nahrávat. Registrační mód je možné aktivovat v uživatelském rozhraní. Pokud mód není aktivovaný, nadřazený systém odmítne všechny požadavky na registraci zasláné tímto způsobem.

3.2.1.2 API klíče

Podřízené systému se při zasílání událostí přes API prokazují klíčem. Ten slouží jednak k zamezení příjmu událostí od neautorizovaných systémů, jednak k identifikaci původu události (zdrojové garáže) v rámci nadřazeného systému. Podřízený systém tedy nemusí znát `id` garáže, ale jen `api_key`.

Vzhledem k těmto požadavkům nelze použít jeden univerzální, ale je nutné pro každý registrovaný podřízený systém vygenerovat unikátní klíč. Ten je pak spolu s dalšími záznamy o garáži uložen v databázi systému. K vytváření klíčů jsem se rozhodl použít systém UUID, umožňující generování náhodných klíčů délky 128 bitů, které jsou (pro praktické účely) unikátní [35].

Klíče je možné v uživatelském rozhraní zneplatnit, a tím odeprít přístup zvolenému podřízenému systému. Tuto operaci lze provést dvěma způsoby. V prvním případě lze smazat z databáze celý záznam příslušné garáže. Tím dojde k zneplatnění jejího klíče, ale také k odstranění zaznamenaných událostí.

Pokud si uživatel přeje data o událostech uchovat, může pouze vygenerovat nový API klíč. Přepsáním klíče se opět zneplatní přístup podřízeného systému (který má stále starý klíč), ale uchovají se zaznamenaná data.

Nově vygenerovaný klíč pak uživatel může nahrát na jiný podřízený systém, a tím například nahradit odcizené monitorovací zařízení. V tomto případě nelze pro nahrání klíče použít registrační mód nadřazeného systému. Ten totiž vždy počítá s vytvořením nové garáže.

Vygenerované klíče jsou v databázi uloženy v čitelné podobě. Klíče by bylo možné před uložením *hashovat*, čímž by při úniku databáze nedošlo k jejich prozrazení. V tom případě by byl klíč v čitelné podobě v uživatelském rozhraní zobrazen pouze jednou, při vytvoření nové garáže. Dále už by byl uchováván jeho *hash*.

Pro ukládání klíčů v čitelné podobě jsem se rozhodl především z důvodů snadného ladění při implementaci nadřazeného a podřízených systémů. Funkci *hashování* klíčů by v případě potřeby neměl být problém doplnit.

3.2.2 Událost – Event

Třída **Event** představuje událost zaznamenanou podřízeným systémem. Jak bylo zmíněno v sekci 3.1, jsou tyto události vázány ke konkrétním garážím, kdy každá událost má jednoznačně určeného původce, a každá garáž libovolné množství událostí.

Nadřazený systém rozlišuje dva základní druhy událostí. Kontrolní (plánované) události slouží ke kontrole funkčnosti podřízených systémů. Tyto události jsou odesílány v pravidelném intervalu, určeném nadřazeným systémem. Ten v odpovědi na požadavek s kontrolní událostí zašle očekávaný čas (počet minut) do dalšího hlášení.

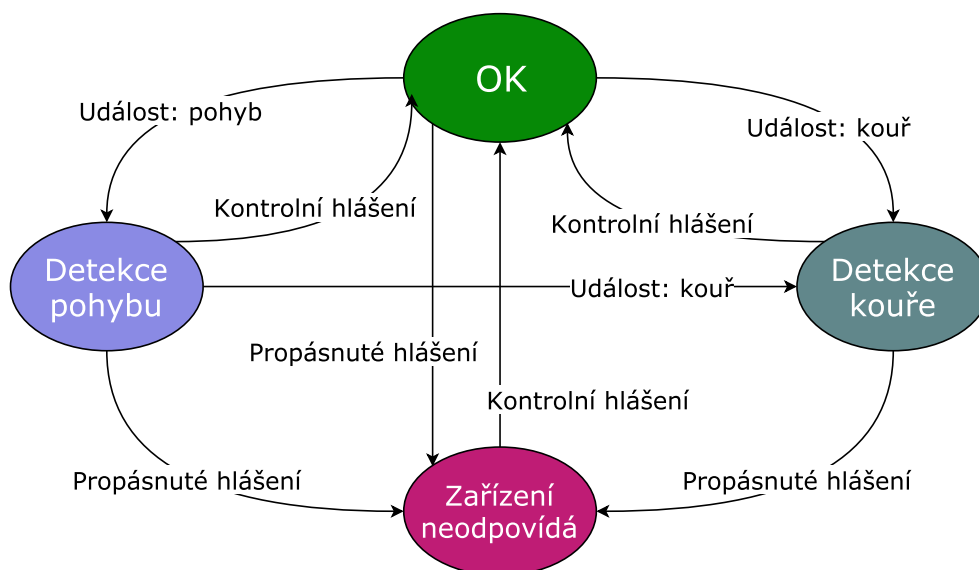
Kromě kontrolních hlášení mohou podřízené systémy vytvářet mimořádné události. Mimořádná událost nastane při překročení mezních hodnot některého z čidel (tedy detekce kouře, pohybu či otevření/zavření dveří).

Třída **Event** obsahuje tyto údaje:

- **id** – identifikace entity v databázi.
- **timestamp** – časové razítko.
- **type** – typ zaznamenané události. Nadřazený systém rozeznává následující typy:
 - **report** – kontrolní hlášení.
 - **door_open** – otevření dveří.
 - **door_close** – zavření dveří.
 - **movement** – detekce pohybu.
 - **smoke** – detekce kouře.
- Garáž, která je původcem události.

3.2.2.1 Vyhodnocení události

Příchozí události a stav garáže jsou nadřazeným systémem vyhodnocovány na základě diagramu na obrázku 3.2. Změna stavu nastává buď v reakci na příchozí událost (například detekci kouře) nebo při pravidelné kontrole zaslaných hlášení.



Obrázek 3.2: Diagram vyhodnocování stavu garáže. Nadřazený systém reaguje na příchozí události odpovídající změnou stavu garáže. Změna stavu může být také vyvolána při pravidelné kontrole propásnutých hlášení (přechod do stavu „Nehlásí se“)

Pokud při této kontrole nadřazený systém zjistí, že neobdržel očekávané kontrolní hlášení, změní stav garáže na „Nehlásí se“. Tato změna nastane vždy, bez ohledu na předchozí stav. Ze stavu „Nehlásí se“ se lze dostat pouze zasláním kontrolního hlášení.

Kontrolní hlášení funguje jako potvrzení, že jakákoliv nenadálá situace v garáži byla vyřešena. Pokud například podřízený systém detekuje kouř, odešle příslušnou událost a pozdrží další kontrolní hlášení až do doby, kdy bude zdroj kouře odstraněn. Poté může zasláním kontrolního hlášení změnit stav garáže zpět na „OK“.

Kromě událostí měnících stav garáže nadřazený systém ještě rozeznává události otevření a zavření dveří garáže. Tyto události nemění stav garáže, pouze aktualizují stav dveří. Nejsou tedy součástí tohoto vyhodnocování.

3.2.2.2 Zasílání upozornění

Na základě příchozích událostí odesílá nadřazený systém notifikační e-maily na adresu zvolenou uživatelem. E-mail s příslušnou událostí je odeslán vždy při změně stavu garáže (mimo změny do stavu „OK“). Tím je zamezeno zbytečnému zasílání duplicitních e-mailů.

3.3 View

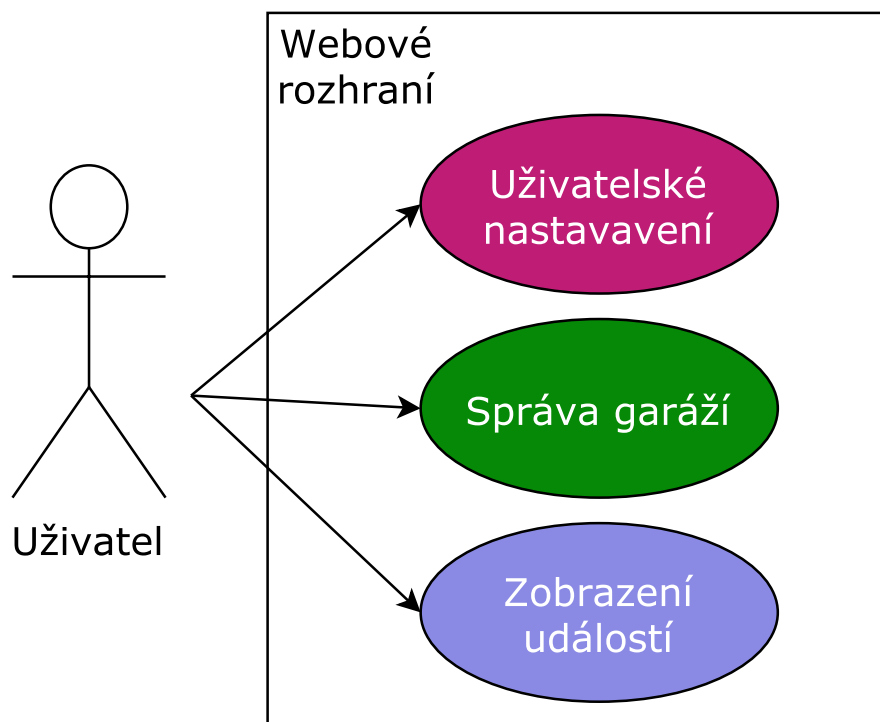
Hlavním *view* nadřazeného systému je webové uživatelské rozhraní. Zde se zobrazuje celkový stav systému, jednotlivých garáží, a zachycené události. Také jsou ze přístupné ovládací prvky pro správu systému. Návrhem tohoto rozhraní se zabývá následující sekce 3.3.1.

Další *view* aplikace představují odpovědi (například vygenerované API klíče) nadřazeného systému podřízeným systému při zpracování jejich požadků. Tento *view* je popsán společně s API systému v sekci 3.4.1.

3.3.1 Webové rozhraní

Webové rozhraní představuje prostředí umožňující uživatel správu celého nadřazeného systému. Rozhraní tvoří hlavní stránka, stránky jednotlivých garáží a stránky uživatelského nastavení (změna hesla a nastavení notifikací).

Případy užití rozhraní se dají shrnout do tří kategorií, popsanych na obrázku 3.3.

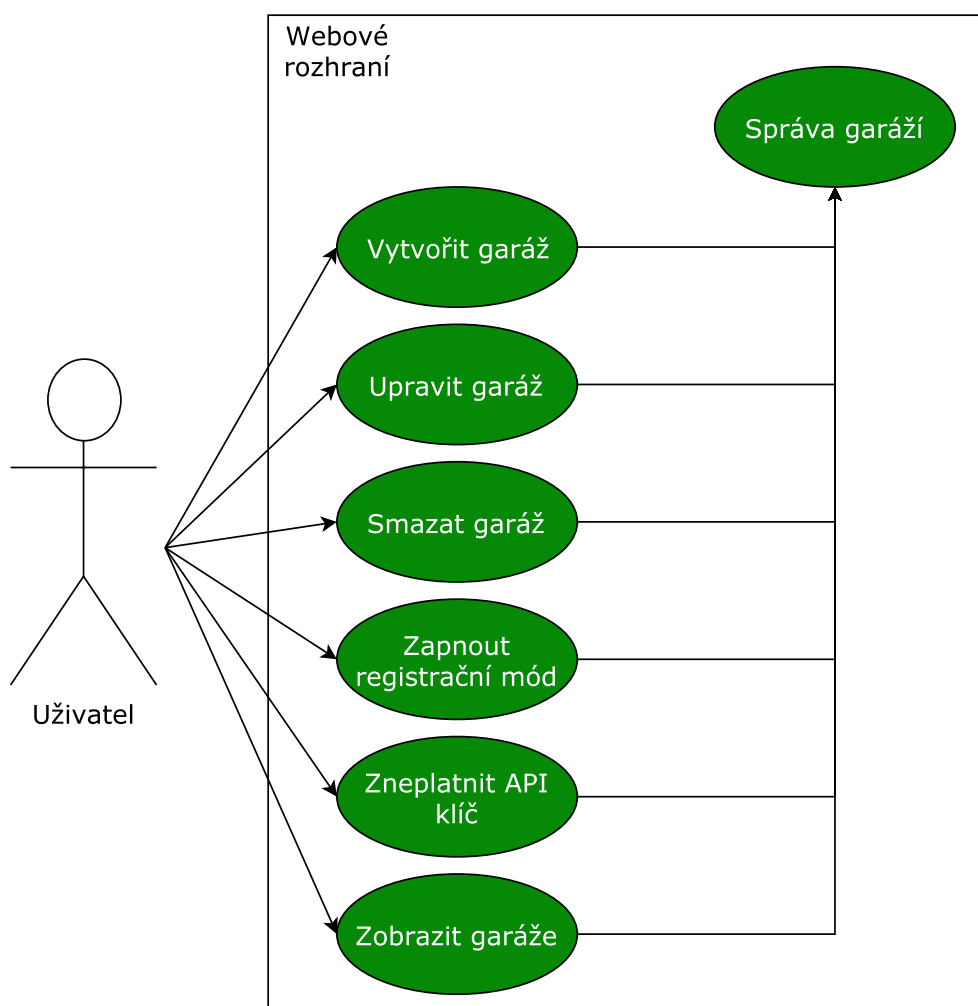


Obrázek 3.3: Vrchní úroveň případů užití webového rozhraní. Uživatel do aplikace přistupuje buď za účelem změny uživatelského nastavení, správy jednotlivý podřízených systémů, případně kvůli kontrole zaznamenaných událostí

3.3.1.1 Správa garáží

Správu garáží tvoří případy užití popsané na obrázku 3.4. Operace vytvoření garáže a zapnutí registračního módu nejsou vázané k žádné konkrétní garáži, a jsou tedy k dispozici na hlavní stránce webového rozhraní ihned po přihlášení.

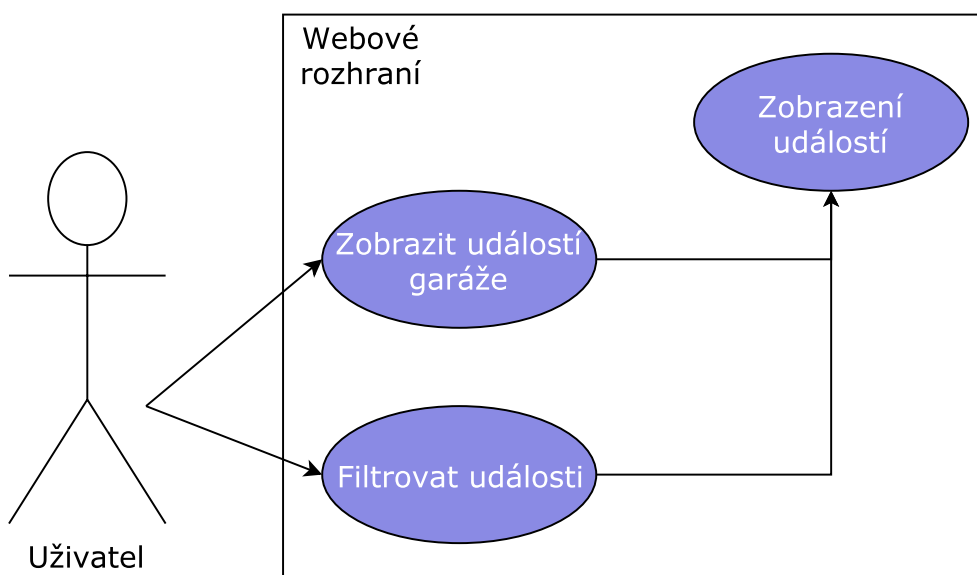
Na hlavní stránce je také zobrazen seznam všech sledovaných garáží včetně jejich stavu, posledního hlášení, dalšího plánovaného hlášení a odkazu na stránku garáže, která obsahuje zbylé možnosti, vázané ke konkrétní garáži.



Obrázek 3.4: Případy užití správy garáží. Zde uživatel zobrazuje stav garáží a jednotlivé garáže spravuje pomocí uvedených operací

3.3.1.2 Zobrazení událostí

Případy užití z kategorie zobrazení událostí popisuje obrázek 3.5. Na stránce garáže může uživatel zobrazit s ní spojené události. Seznam událostí pak může filtrovat podle typu (viz sekci 3.2.2).



Obrázek 3.5: Případy užití zobrazení událostí. Uživatel zobrazuje události spojené s konkrétní garáží. Zobrazené události filtruje podle jejich typu

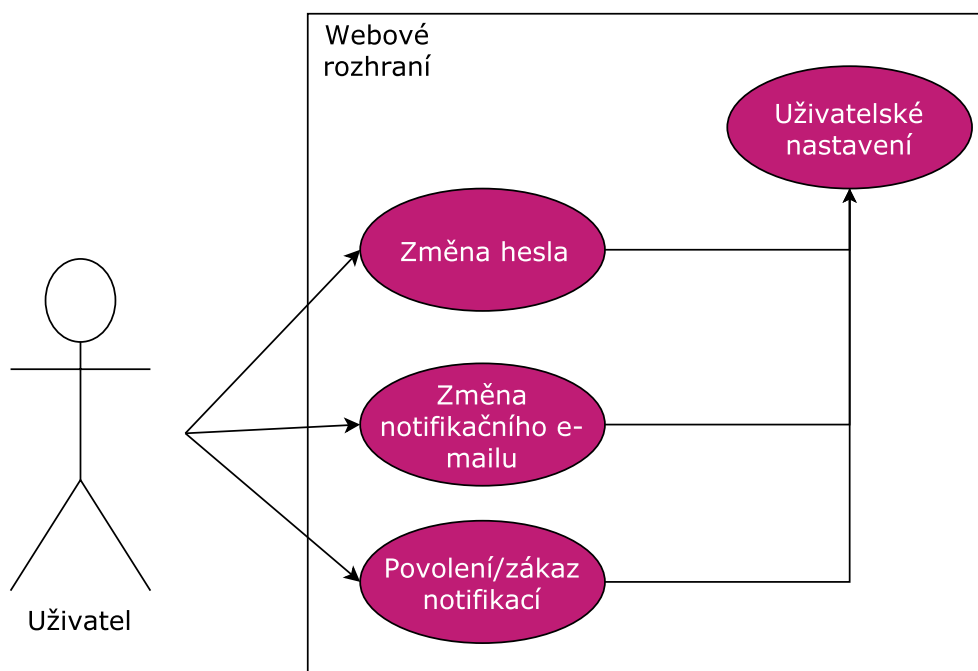
3.3.1.3 Uživatelské nastavení

Uživatelské nastavení není uloženo přímo v databázi, ale zvláště v konfiguračním souboru `user_settings.ini`. Díky tomu je snazší toto nastavení v případě potřeby změnit i bez nutnosti otvírat webové rozhraní.

3.4 *Controller*

Controller slouží ke zpracování požadavků uživatele a podřízených systémů. Jelikož jde o HTTP požadavky, definuje *controller* URL, přiřazuje k nim akce, které se mají provést (kontrola přihlášení, přesměrování atd.) a vrací příslušné odpovědi (například vygenerované stránky).

Controller aplikace se dá rozdělit na dvě části. První část zpracovává požadavky zaslané uživatelem z webového rozhraní. Propojuje tedy případy užití popsané v sekci 3.3.1 s operacemi na modelu ze sekce 3.2. URL definovaná v této části jsou využívána pouze uvnitř aplikace, a nejsou součástí API nadřazeného systému.



Obrázek 3.6: Případy užití uživatelského nastavení. Uživatel ve webovém rozhraní provádí změnu hesla, změnu notificačního e-mailu, případně zákaz/povolení notifikací

Druhá část zpracovává požadavky podřízených systémů. Zde definovaná URL tvoří API nadřazeného systému, pomocí kterého se mohou podřízené systémy registrovat a nahrávat nové události. Popisem API, včetně formátu požadavků a návratových hodnot, se zabývá sekce 3.4.1.

3.4.1 API

API nadřazeného systému slouží ke komunikaci s podřízenými systémy. S jeho pomocí se podřízený systém může registrovat a začít nadřazenému systému posílat zaznamenané události.

Odpovědi nadřazeného systému na příchozí požadavky jsou ve formátu JSON. Každá odpověď obsahuje návratový kód a případně další data, která je nutné zaslat podřízenému systému.

3.4.1.1 Registrační požadavek

Registrační požadavek slouží k zaregistrování nového podřízeného systému a vytvoření záznamu s ním spojené garáže. Hlavním účelem tohoto požadavku je snadná distribuce API klíče na podřízené systémy – klíč není nutné nahrávat

ručně, ale je odeslán v odpovědi na tento požadavek (pro bližší informace o klíčích viz sekci 3.2.1.2).

Kvůli bezpečnosti je tento požadavek povolen pouze pokud je v uživatelském rozhraní zapnutý registrační mód. Pokud je mód vypnutý, odpovídá nadřazený systém na všechny registrační požadavky chybovým kódem *403* – *Forbidden*. Tím je částečně zabráněno registraci nežádoucích podřízených systémů.

Registrační požadavek má následující formát (zaslání registračního požadavku v jazyce Python je popsáno v ukázce 2):

- URL – `/api/garages`.
- HTTP metoda – POST.
- Návrátová hodnota – JSON:
 - `status` – návratový kód (201 při vytvoření garáže, 403 při vypnutém reg. módu).
 - `api_key` – unikátní klíč k API nadřazeného systému (pouze v případě úspěchu, tedy vytvoření garáže).

```
import requests
# server nadřazeného systému je přístupný
# na adrese raspberrypi.local
url = 'https://raspberrypi.local/api/garages'

# požadavek 'post', bez ověření
# self-signed certifikátu
r = requests.post(url, verify=False)

if r.status == 201:
    api_key = r.json()['api_key']
```

Ukázka 2: Zaslání registračního požadavku nadřazenému systému provozovaném na adrese `raspberrypi.local`

3.4.1.2 Kontrolní hlášení

Kontrolní hlášení slouží k ověření činnosti podřízeného systému. Jeho zasláním podřízený systém také potvrzuje, že stav sledované garáže je v pořádku.

Požadavek zasílající kontrolní hlášení má tento formát:

- URL – `/api/report_event`.
- HTTP metoda – POST.

3. NÁVRH

- Hlavička požadavku:
 - `api_key` – unikátní klíč k API nadřazeného systému.
- Návrátová hodnota – JSON:
 - `status` – návratový kód (201 při vytvoření hlášení, 403 při neplatném klíči).
 - `period` – počet minut do dalšího očekávaného hlášení.

```
import requests
# server nadřazeného systému je přístupný
# na adrese raspberrypi.local
url = 'https://raspberrypi.local/api/report_event'

# API klíč (získaný například reg. požadavkem)
# je zasílán v~hlavičce požadavku
headers = {'api_key' : api_key}

# požadavek 'post', bez ověření
# self-signed certifikátu
r = requests.post(url, headers=headers, verify=False)

if r.status == 201:
    #odpověď -- čas do dalšího hlášení
    period = r.json()['period']
```

Ukázka 3: Zaslání kontrolního hlášení nadřazenému systému provozovaném na adrese `raspberrypi.local`

3.4.1.3 Ostatní události

Vytváření dalších událostí je stejné jako vytváření kontrolního hlášení. Liší se pouze použité URL. Odpovědi na tyto požadavky neobsahují žádná data od nadřazeného systému, pouze návratový kód informující o úspěchu či chybě:

- URL – podle typu události:
 - `/api/door_open_event`
 - `/api/door_close_event`
 - `/api/movement_event`
 - `/api/smoke_event`
- HTTP metoda – POST.

- Hlavička požadavku:
 - `api_key` – unikátní klíč k API nadřazeného systému.
- Návrátová hodnota – JSON:
 - `status` – návratový kód (201 při vytvoření události, 403 při neplatném klíči).

```
import requests

# stejné jako při zasílání kontrolního hlášení
# liší se pouze url
url = 'https://raspberrypi.local/api/smoke_event'
headers = {'api_key' : api_key}

r = requests.post(url, headers=headers, verify=False)

if r.status == 201:
    #odpověď neobsahuje žádná data
    ...
```

Ukázka 4: Zaslání události detekce kouře nadřazenému systému provozovaném na adrese `raspberrypi.local`

3.5 Autentizace uživatele

Do webového rozhraní aplikace je povolen pouze autorizovaný přístup, a je tedy nutné nějakým způsobem ověřit identitu uživatele. Jelikož aplikace nepočítá s odlišnými stupni přístupu (a tedy každý uživatel s přístupem do rozhraní má přístup ke všem jeho částem), rozhodl jsem se, že nebudu vytvářet infrastrukturu uživatelských účtů, protože by měla pouze omezené využití.

Přístup do rozhraní tedy nevyžaduje vytvoření účtu a zadávání uživatelského jména, ale pouze heslo. *Hash* tohoto hesla (doplněného solí), vytvořený pomocí algoritmu Bcrypt, je uložen v souboru s uživatelským nastavením aplikace.

Hashování hesla je zde použito především pro případ, že by uživatel použil stejné heslo jaké používá v jiných aplikacích či službách. V případě prozrazení uživatelských dat (například při odcizení systému) tedy útočník nemůže z otisku zjistit původní použité heslo, tudíž přístup k těmto uživatelským účtům není kompromitován.

Stejně tak někdo s přístupem k serveru, na kterém nadřazený systém běží, nemá automaticky přístup do webového rozhraní. Nicméně má stále přístup k celé (nešifované) databázi a také může heslo libovolně měnit (spočítáním

nového otisku), tedy z praktického hlediska je přístup k serveru téměř ekvivalentní s přístupem do webového rozhraní.

3.5.1 První přihlášení

Při prvním startu nadřazeného systému je zvoleno jednoduché implicitní heslo (*password*), kterým se uživatel může přihlásit do webového rozhraní.

Pokud nadřazený systém detekuje použití tohoto hesla, po přihášení uživatele ihned přesměruje na stránku umožňující změnu hesla a zobrazí příslušné varování.

Implementace

V této kapitole je popsána implementace nadřazeného systému podle návrhu z kapitoly 3. Jednotlivé sekce tedy popisují implementační detaily dílčích částí návrhového vzoru MVC. Poslední sekce 4.5 se zabývá implementací autentizace uživatele.

4.1 Struktura aplikace

Aplikace nadřazeného systému je rozdělena do tří modulů:

- **mod_main** – hlavní modul aplikace. V tomto modulu je implementován *model* systému popsáný v sekci 3.2. *Model* je využíván i ostatními moduly (konkrétně modulem **mod_api**). Dále tento modul implementuje webové rozhraní správy nadřazeného systému.
- **mod_api** – modul implementující API systému. Zde je implementován *controller* ze sekce 3.4.1, definující URL, pomocí kterých mohou podřízené systémy komunikovat s nadřazeným systémem.
- **mod_auth** – modul implementující autentizaci uživatele při přihlašování do webového rozhraní.

K rozdělení aplikace na jednotlivé moduly je použit nástroj frameworku Flask nazvaný *blueprints* [36]. Hlavní důvod k využití modulů je oddělení částí aplikace podle jejich funkce.

Při strukturování aplikace jsem vycházel z článku *How To Structure Large Flask Applications* [37].

4.2 Implementace *modelu*

Jak je zmíněno v sekci 3.2, *model* aplikace je tvořen třídami **Garage** a **Event**. Tyto třídy přímo využívají databázi nadřazeného systému, k jejich implemen-

taci je proto použit framework SQLAlchemy [38], který výrazně usnadní práci s databází.

Tento framework poskytuje přístup k SQL databázím přímo z jazyka Python, takže není nutné psát prakticky žádný SQL kód. Tabulku v databázi je možné definovat jako Python třídu s příslušnými atributy a SQLAlchemy vytvoří odpovídající databázové schéma.

SQLAlchemy také umožňuje snadno definovat databázové vztahy. V *modelu* nadřazeného systému se vyskytuje pouze vztah 1 : *N* mezi třídami **Garage** a **Event** (jedna garáž má mnoho událostí, každá událost má právě jednu garáž). Tento vztah je možno pomocí SQLAlchemy definovat následujícím způsobem:

```
from sqlalchemy import Column, ForeignKey, Integer
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

# třídy Event a Garage
# dědí SQLAlchemy metody
# třídy Base
class Event(Base):
    ...
    # odkaz na příslušnou garáž
    garage_id = Column(Integer, ForeignKey(
        'Garage.id'), nullable=False)

class Garage(Base):
    ...
    # definice 1:N vztahu mezi garáží a událostí
    events = relationship('Event', backref='Garage')
```

Ukázka 5: Vytvoření vztahu 1 : *N* mezi třídami **Garage** a **Event**

4.2.1 Kontrola zmeškaných hlášení

V databázi nadřazeného systému je potřeba pravidelně provádět kontrolu, zda podřízené systémy zaslaly v očekávaný čas kontrolní hlášení. Pokud bylo plánované hlášení promeškáno, je nutné změnit stav příslušné garáže na „Nehlásí se“.

Aplikace nadřazeného systému nemá v zásadě možnost, jak tuto kontrolu sama iniciovat, neboť pouze reaguje na příchozí požadavky (od uživatele či podřízeného systému). Provedení kontroly může být důsledkem takového požadavku, například pokud uživatel otevře hlavní stránku webového rozhraní.

Provádět kontrolu hlášení pouze v reakci na vnější podnět však není dostatečné. Pokud by nadřazený systém musel pro provedení kontroly hlášení čekat interakci s webovým rozhraním (nebo například s podřízeným systémem), nemuselo by vůbec dojít ke změně stavu garáže a tedy ani k odeslání notifikačního e-mailu. Je tedy nutné zajistit pravidelné provádění kontrol na základě vnitřního podnětu.

Tento problém jsem vyřešil použitím plánovače APScheduler [39]. APScheduler funguje jako knihovna do Pythonu, a umožňuje plánovat provádění zvolených funkcí. Nejde tedy o externí program, plánovač je přímo součástí kódu nadřazeného systému [39]. Vytvoření pravidelné kontroly hlášení podřízených systémů lze implementovat tímto způsobem:

```
from apscheduler.schedulers.background import \
    BackgroundScheduler

# BackgroundScheduler běží v~samostatném vlákně,
# neblokuje tedy webovou aplikaci
scheduler = BackgroundScheduler()

# přidání pravidelného úkolu do plánovače
# Garage.check_reports je statická metoda
# třídy Garage, která provede kontrolu hlášení
# u~všech garáží v~databázi (a případně upraví jejich stav)
scheduler.add_job(Garage.check_reports, 'interval', minutes=5)
scheduler.start()
```

Ukázka 6: Pravidelná kontrola hlášení podřízených systémů pomocí knihovny APScheduler. Po startu plánovače je každých 5 minut volána metoda `Garage.check_reports()`

Plánovač se kromě kontroly hlášení hodí také při vypínání registračního módu. Ten z bezpečnostních důvodů po aktivaci běží po dobu tří minut. Jeho vypnutí je naplánováno obdobně jako kontrola hlášení, jediný rozdíl je, že úkol není spouštěn v pravidelném intervalu, ale pouze jednou.

4.2.2 Zasíláním upozornění

4.3 Implementace *view*

Tato část se věnuje implementaci stránek webového rozhraní nadřazeného systému. Jednotlivé HTML stránky rozhraní jsou generovány pomocí šablonovacího systému Jinja [40], který je součástí frameworku Flask [41].

Jinja umožňuje dynamické generování HTML stránek z předem vytvořených šablon a dalších vstupů, v tomto případě především dat z *modelu* aplikace. Tato data jsou šabloně předána *controllerem* (viz sekci 4.4). V šablonách

Jinja používá syntaxi založenou na Pythonu [40]. Například HTML šablona pro tabulku se zaznamenanými událostmi může vypadat následovně:

```
<table class=basic_table>
  <tr>
    <th>Seznam událostí</th>
  </tr>
  <!-- použití cyklu v~systému Jinja -->
  {% for event in garage.events %}
    <tr>
      <!-- přístup k~proměnné pomocí bloku {{ }} -->
      <td class=event_type_{{event.type}}>
        {{ event | event_filter }}
      </td>
    </tr>
  {% endfor %} <!--konec bloku cyklu -->
</table>
```

Ukázka 7: HTML šablona tabulky zaznamenaných událostí, využívající šablonovací systém Jinja. Proměnná `garage` je šabloně předána *controllerem* aplikace

Jinja také dovoluje definici filtrů, sloužících ke zpracování zobrazovaných dat [40]. V ukázce 7 je použit filtr `event_filter`. Ten dostane jako vstup instanci třídy `event` a vytvoří její textovou reprezentaci s údaji o typu a čase události. Tím je oddělena implementace této třídy v *modelu* a její zobrazení ve *view*.

4.3.1 Formuláře

4.3.2 Filtrování událostí

4.4 Implementace *controlleru*

4.4.1 API *controller*

4.5 Autentizace uživatele

Testování

Závěr

co sem udelal

- analiza
 - definice nadrazeného systému, definice požadavku na něj (specifikace), popis podřízeného systému, definice události jako základní komunikační jednotky, bezpečnost a spolehlivost systému
 - vyber kom. protokolu, výhody a nevýhody https, mqtt, volba certifikátu (delame prototyp), další možnosti provozu (cloud), komunikační modely client/server, publisher/subscriber
 - autentizace podřízených systému, api klice
 - problem sirení api klicu na subsystemy – reg mod
 - vyber dalšího softwaru, programovacího jazyku, frameworku (pochvalit framework flask, pohodová implementace)
 - vyber uložiste (db vs logy, persistence), resil sem: systemové nároky, rust velikosti
 - notifikace, vyber způsobu zaslání (email), nevhodnost provozu vlastního serveru, vyber služby pro email
 - vyber platformy, podporovaný sw, rpi vs zybo, zybo zajímavý ale moc drahý, zmínit rtc
- návrh
 - návrh systému, použití mvc, možnosti dalších controlleru (mqtt) – rozširitelnost
 - stav garazi, fsm, přechody stavu, notifikace při změně stavu – co to resi za problémy (casty maily), jak resit přechod zpět do stavu OK napr po požaru
 - api systému, návrh, přidávání události, registrační mod

- uživatelsky rozhraní, přihlasování, uživatelsky nastavení
 - jak jsem řešil problém s autorizací přístupu k posílání mailů
- implementace
 - spuštění použitéch technologií/frameworků
 - z nich zajímavější flask, jinja, apscheduler
- sepsání uživatelské příručky

Literatura

- [1] Apache Software Foundation: Apache – Frequently Asked Questions. 2015, [cit. 2018-03-04]. Dostupné z: <https://wiki.apache.org/httpd/FAQ>
- [2] Arduino: Web Client. 2015, [cit. 2017-10-25]. Dostupné z: <https://www.arduino.cc/en/Tutorial/WebClient>
- [3] Grokhotkov, I.: esp8266wifi – Client Example. 2017, [cit. 2017-10-25]. Dostupné z: <https://github.com/esp8266/Arduino/blob/master/doc/esp8266wifi/client-examples.rst>
- [4] Electronic Frontier Foundation: Let’s Encrypt – Frequently Asked Questions. 2017, [cit. 2017-11-07]. Dostupné z: <https://letsencrypt.org/docs/faq/>
- [5] Electronic Frontier Foundation: Certbot – About. [cit. 2017-10-18]. Dostupné z: <https://certbot.eff.org/about/>
- [6] Wallen, J.: When are self-signed certificates acceptable for businesses? 2017, [cit. 2017-11-08]. Dostupné z: <https://www.techrepublic.com/article/when-are-self-signed-certificates-acceptable-for-businesses/>
- [7] Reitz, K.: Requests: HTTP for Humans. 2018, [cit. 2018-03-01]. Dostupné z: <http://docs.python-requests.org/en/master/>
- [8] The Python Software Foundation: The Python Tutorial. 2018, [cit. 2018-03-04]. Dostupné z: <http://flask.pocoo.org/>
- [9] Arduino: EEPROM Library. 2018, [cit. 2017-03-01]. Dostupné z: <https://www.arduino.cc/en/Reference/EEPROM>

- [10] Lampkin, V.: What is MQTT and how does it work with WebSphere MQ? 2012, [cit. 2017-10-25]. Dostupné z: https://www.ibm.com/developerworks/mydeveloperworks/blogs/aimsupport/entry/what_is_mqtt_and_how_does_it_work_with_websphere_mq?lang=en
- [11] Jaffey, T.: MQTT and CoAP, IoT Protocols. 2014, [cit. 2017-11-12]. Dostupné z: https://eclipse.org/community/eclipse_newsletter/2014/february/article2.php
- [12] Newsom, C.: Mosquitto Message Broker. 2016, [cit. 2017-11-16]. Dostupné z: <https://github.com/mqtt/mqtt.github.io/wiki/Mosquitto-Message-Broker>
- [13] Light, R.: mosquitto.tls – Mosquitto Manual. [cit. 2017-11-20]. Dostupné z: <https://mosquitto.org/man/mosquitto-tls-7.html>
- [14] Nguyen, J.: OpenSSL Certificate Authority. 2015, [cit. 2017-11-20]. Dostupné z: <https://jamielinux.com/docs/openssl-certificate-authority/>
- [15] Light, R.: mosquitto.conf – Mosquitto Manual. [cit. 2017-11-20]. Dostupné z: <https://mosquitto.org/man/mosquitto-conf-5.html>
- [16] Light, R.: libmosquitto – Mosquitto Manual. [cit. 2017-11-21]. Dostupné z: <https://mosquitto.org/man/libmosquitto-3.html>
- [17] Eclipse Foundation: Python – Mosquitto Documentation. [cit. 2017-11-21]. Dostupné z: <https://mosquitto.org/documentation/python/>
- [18] Eclipse Foundation: Embedded MQTT C/C++ Client Libraries. [cit. 2017-11-21]. Dostupné z: <http://www.eclipse.org/paho/clients/c/embedded/>
- [19] Eclipse Foundation: Paho C++ Documentation. [cit. 2017-11-21]. Dostupné z: <http://www.eclipse.org/paho/files/cppdoc/index.html>
- [20] Eclipse Foundation: Paho Python Documentation. [cit. 2017-11-21]. Dostupné z: <https://pypi.python.org/pypi/paho-mqtt>
- [21] Hipp, D. R.: About SQLite. [cit. 2017-12-12]. Dostupné z: <https://www.sqlite.org/about.html>
- [22] Hipp, D. R.: Datatypes In SQLite Version 3. [cit. 2018-03-03]. Dostupné z: <https://www.sqlite.org/datatype3.html>
- [23] Anicas, M.: Why You May Not Want To Run Your Own Mail Server. 2014, [cit. 2018-02-26]. Dostupné z: <https://www.digitalocean.com/community/tutorials/why-you-may-not-want-to-run-your-own-mail-server>

-
- [24] Ronacher, A.: Flask. 2018, [cit. 2018-03-04]. Dostupné z: <https://docs.python.org/3/tutorial/index.html>
 - [25] Django Software Foundation : Why Django? 2018, [cit. 2018-03-04]. Dostupné z: <https://www.djangoproject.com/start/overview/>
 - [26] Benchoff, B.: Introducing the Raspberry Pi 3. 2016, [cit. 2018-01-25]. Dostupné z: <https://hackaday.com/2016/02/28/introducing-the-raspberry-pi-3/>
 - [27] Adafruit: Adding a Real Time Clock to Raspberry Pi. 2016, [cit. 2018-01-25]. Dostupné z: <https://learn.adafruit.com/adding-a-real-time-clock-to-raspberry-pi/overview>
 - [28] Raspberry Pi Foundation : Raspbian FAQ. [cit. 2018-03-04]. Dostupné z: <https://www.raspbian.org/RaspbianFAQ>
 - [29] Digilent: ZYBO FPGA Board Reference Manual. 2016, [cit. 2018-01-29]. Dostupné z: https://reference.digilentinc.com/_media/zybo:zybo_rm.pdf
 - [30] Teich, J.: Hardware/Software Codesign: The Past, the Present, and Predicting the Future. *Proceedings of the IEEE*, ročník 100, 5 2012: s. 1411 – 1430. Dostupné z: <http://ieeexplore.ieee.org/document/6172642/>
 - [31] Xilinx: Getting Started with Xilinx for Zynq-7000. [cit. 2018-01-29]. Dostupné z: http://xillybus.com/downloads/doc/xillybus_getting_started_zynq.pdf
 - [32] Digilent: Pmod RTCC Reference Manual. 2016, [cit. 2018-01-30]. Dostupné z: https://reference.digilentinc.com/_media/reference/pmod/pmodrtcc/pmodrtcc_rm.pdf
 - [33] DigitalOcean: DigitalOcean – Pricing. 2018, [cit. 2018-02-05]. Dostupné z: <https://www.digitalocean.com/pricing/>
 - [34] DigitalOcean: DigitalOcean – Droplets. 2018, [cit. 2018-02-05]. Dostupné z: <https://www.digitalocean.com/products/droplets/>
 - [35] Leach, P.; aj.: A Universally Unique Identifier (UUID) URN Namespace. RFC 4122, RFC Editor, July 2005. Dostupné z: <https://www.rfc-editor.org/rfc/rfc4122.txt>
 - [36] Ronacher, A.: Modular Applications with Blueprints. 2017, [cit. 2018-03-15]. Dostupné z: <http://flask.pocoo.org/docs/0.12/blueprints/>
 - [37] Tezer, O. S.: How To Structure Large Flask Applications. 2014, [cit. 2018-03-16]. Dostupné z: <https://www.digitalocean.com/community/tutorials/how-to-structure-large-flask-applications>

LITERATURA

- [38] Bayer, M.: SQLAlchemy. [cit. 2018-03-16]. Dostupné z: <https://www.sqlalchemy.org/>
- [39] Grönholm, A.: Advanced Python Scheduler. 2014, [cit. 2018-03-17]. Dostupné z: <https://apscheduler.readthedocs.io/en/latest/index.html>
- [40] Ronacher, A.: Jinja2 Documentation. 2008, [cit. 2018-03-17]. Dostupné z: <http://jinja.pocoo.org/docs/2.10/>
- [41] Ronacher, A.: Templates. 2017, [cit. 2018-03-17]. Dostupné z: <http://flask.pocoo.org/docs/0.12/templating/>

Nasazení

tady bude něco vo nasazování na RPI (tj rozjet apache, vygenerovat certifikáty atd., viz <https://github.com/ggljzr/mi-dip-impl/tree/master/deployment>)

tu databázi je potřeba nastavit práva/dat nekam aby do ni ten apache mohl zapisovat.

Uživatelská příručka

Tato uživatelská příručka se zabývá webovým rozhraním nadřazeného systému. Sekce příručky odpovídají stránkám jednotlivým stránkám rozhraní. V každé sekci jsou popsány možnosti, které má uživatel na dané stránce.

Přihlášení do webového rozhraní je prováděno pomocí hesla. Přihlašovací formulář je zobrazen na obrázku B.1.

Správa garáží



The image shows a dark-themed login form. At the top, the title 'Přihlášení' is centered. Below it, the label 'Heslo:' is followed by a light gray rectangular input field. At the bottom of the form is a button labeled 'Přihlásit'.

Obrázek B.1: Přihlašovací formulář webového rozhraní

B.1 První přihlášení

Pro první přihlášení je možné použít implicitní heslo *password*. Poté je uživatel přesměrován na formulář se změnou hesla a vyzván ke změně z implicitní hodnoty (viz obrázek B.2).

Změna hesla

Je nutné změnit heslo z implicitně nastavené hodnoty

Staré heslo:

Nové heslo:

Nové heslo znovu:

Potvrdit

Obrázek B.2: Výzva ke změně hesla

B.2 Hlavní stránka

Menu		Garáže				
Nová garáž	1					
Registrační mód:						
Vypnutý	2					
Zapnout reg. mód						
Nastavení	3					
Změna hesla						
Uživatel						
Odhlásit						

ID	Označení	Dveře	Poslední hlášení	Další plánované hlášení	Stav
[1]	Dolní garáž	Otevřeno	2018-03-08 16:44:02.234987	2018-03-08 16:45:02.234987	Nehlásí se
[2]	Druhá garáž	Otevřeno	Žádné	Žádné	OK

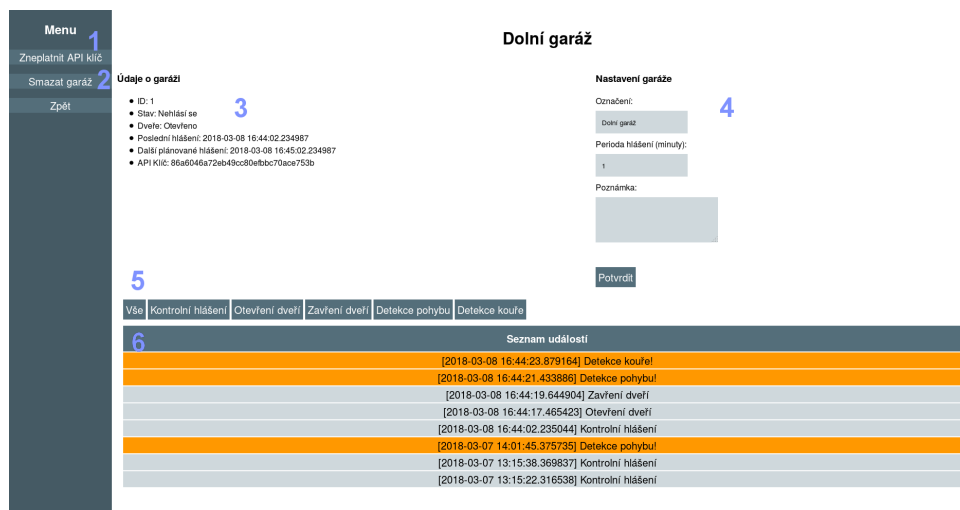
Obrázek B.3: Hlavní stránka aplikace

Po úspěšném přihlášení se zobrazí hlavní stránka webového rozhraní. Zde má uživatel k dispozici následující možnosti (viz čísla na obrázku B.3):

1. **Nová garáž** – vytvoření nové garáže pomocí uživatelského rozhraní. Nová garáž je přidána do seznamu garáží na této stránce.

2. **Registrační mód** – indikátor stavu registračního módu. Registrační mód povoluje podřízeným systémům vytvářet nové garáže registračním požadavkem. Při registraci pomocí tohoto módu je podřízenému systému automaticky zaslán API klíč, a není nutné ho ručně nahrávat. Po zapnutí je registrační mód aktivní po dobu tří minut, poté se sám automaticky vypne.
3. **Nastavení** – uživatelské nastavení aplikace:
 - **Změna hesla** – možnost změny přístupového hesla do webového rozhraní.
 - **Uživatel** – viz sekci B.4.
 - **Odhlásit** – odhlášení z webového rozhraní.
4. Seznam sledovaných garáží (podřízených systémů):
 - **ID** – identifikátor garáže v systému, zároveň slouží jako odkaz na stránku garáže (viz sekci B.3).
 - **Označení** – Uživatelem zvolené označení garáže.
 - **Dveře** – Stav dveří garáže (otevřeno/zavřeno).
 - **Poslední hlášení** – Datum a čas posledního zaznamenaného hlášení.
 - **Další plánované hlášení** – Datum a čas dalšího plánovaného hlášení.
 - **Stav** – Stav garáže (OK, Nehlásí se, Detekce pohybu, Detekce kouře).
5. Záznam garáže.

B.3 Stránka garáže



Obrázek B.4: Stránka garáže

Na obrázku B.4 je zobrazena stránka konkrétní garáže. Zde jsou k dispozici tyto možnosti:

1. **Zneplatnit API klíč** – vygeneruje nový API klíč garáže. Po jeho vygenerování již nebude podřízený systém v této garáži moci zasílat nové události. Pro obnovení přístupu je nutné nahrát na systém nový klíč, případně vytvořit novou garáž pomocí registračního módu.
2. **Smazat garáž** – smaže zobrazenou garáž, včetně zaznamenaných událostí.
3. **Údaje o garáži**, včetně API klíče.
4. **Nastavení garáže:**
 - **Označení** – uživatelem zvolené označení, které se zobrazí na hlavní stránce.
 - **Perioda hlášení** – perioda kontrolních hlášení podřízeného systému (v minutách).
 - **Poznámka** – volitelná poznámka ke garáži (max 256 znaků).
5. **Filtry událostí** – možnost filtrovat zobrazené události podle typu.
6. **Události** – tabulka zaznamenaných událostí, včetně data a času.

B.4 Uživatelské nastavení

V uživatelském nastavení může uživatel měnit e-mailovou adresu, na kterou nadřazený systém zasílá upozornění o změně stavu garáže. Může zde také tato upozornění úplně zakázat.

Seznam použitých zkratek

API

AXI

CPU

CSRF

DDR

EEPROM

FPGA

HDMI

HTML

HTTP Graphical user interface

HTTPS Graphical user interface

I2C

IP Ip jadro

JSON

LCD

LPDDR

LTS

MQTT Graphical user interface

MVC

C. SEZNAM POUŽITÝCH ZKRATEK

NTP Network time protocol

OSI

PSK

QoS

RAM

RTC

SD

SSD

SoC

TCP/IP Graphical user interface

URL

USB

UUID

Obsah přiloženého CD

	readme.txt.....	stručný popis obsahu CD
	exe	adresář se spustitelnou formou implementace
	src	
	impl.....	zdrojové kódy implementace
	thesis	zdrojová forma práce ve formátu L ^A T _E X
	text	text práce
	thesis.pdf	text práce ve formátu PDF
	thesis.ps	text práce ve formátu PS