Integračný softvér pre smart home inšpirovaný mikroslužbami
Analýza a návrh riešenia

Patrícia Szepesiová

1Im, 2019-2020

Abstrakt Práca sa zaoberá analýzou možností dekompozície funkcionality monolitických integračných riešení pre „Smart home“. Analyzuje možnosti použitia mikroslužieb v prostredí internetu vecí. Popisuje návrh prototypu integračného softvéru pre „Smart home“ na princípe podobnom mikroslužbám.

Kľúčové slová: internet vecí, integračný softvér, protokol MQTT, mikroslužby

1 Úvod

Oblasť internetu vecí sa ešte stále vyvíja obrovskou rýchlosťou. Počet zariadení pripojených do siete dosahuje desiatky miliárd a naďalej rastie. Na trhu pribúdajú nové zariadenia, ktoré sú vylepšením existujúcich, no tiež také, ktoré prinášajú novú funkcionalitu. Ich ceny sú dostupné, čo takisto pridáva k ich rozšíreniu. Spolu so zariadeniami pribúda aj množstvo nových protokolov a aplikácií.

S rozmachom internetu vecí súvisí aj čoraz obľúbenejšia predstava inteligentných domovov. Možnosť ovládať a kontrolovať celú domácnosť pomocou jednej aplikácie, navyše v čase, keď sa doma nikto nenachádza, má mnoho výhod z hľadiska bezpečnosti, šetrenia finančných prostriedkov, ale aj komfortu. Chytré zariadenia umožňujú regulovať teplotu v jednotlivých izbách, pripojiť či odpojiť jednotlivé elektrické zásuvky a spotrebiče, detegovať únik plynu a mnoho ďalších užitočných funkcií.

Aby sme dokázali všetky senzory a aktuátory kontrolovať, napríklad pomocou mobilného telefónu, potrebujeme na to integračný softvér.

2 Integračné riešenia pre Smart Home

Pod pojmom internet vecí (skrátene IoT - *internet of things*) rozumieme sieť heterogénnych zariadení a softvéru, ktoré spolu komunikujú, a výmenou dát a ich spracovaním spolu vytvárajú nejakú pridanú hodnotu. Hlavnou úlohou integračného softvéru je najmä ovládanie a získavanie dát z jednotlivých koncových

uzlov siete týchto „vecí“, uchovávanie dát do databázy, spracovanie udalostí či webová správa. S rýchlym nárastom IoT oblasti sa vyžaduje súbežne aj vývoj integračných softvérov.

Aby bol vyvíjaný softvér spoľahlivý, bezpečný a stabilný, je potrebné, aby spĺňal určité požadované vlastnosti. Autori článku [6] popisujú skupinu vlastností, ktoré by mal softvér pre IoT spĺňať:

- schopnosť jednotlivých časti systému samostatne sa vyvíjať, nezávisle od ostatných častí systému,
- softvér by mal byť škálovateľný a vývoja-schopný, aby zabezpečil podporu pre stále pribúdajúce nové zariadenia,
- mal by byť ľahko testovateľný,
- dostatočne jednoduchý na to, aby s nim vedeli pracovať a implementovať aj vývojári so základnými znalosťami,
- odolný voči chybám, zlyhanie nejakej časti by nemalo nutne ovplyvniť chod celého systému a ten by sa mal vedieť z chyby zotaviť a pokračovať v činnosti,
- mal by mať jednoduché nasadenie, jednoducho sa inštalovať a odinštalovať, aktivovať a deaktivovať aj aktualizovať,
- schopnosť komunikovať medzi rozličnými doménami a
- zabezpečovať dobrú koordináciu jednotlivých častí systému, ktoré spolu musia spolupracovať, aby vykonali požadovanú akciu.

3 Prehľad existujúcich riešení

V tejto kapitole si popíšeme niekoľko existujúcich riešení integračného softvéru. Ich spoločnou charakteristikou je ich možné nasadenie ako automatizačného riešenia pre „Smart home“.

3.1 Control-Freak

Control-Freak bol vytvorený na správu, programovanie a automatizáciu zariadení. Aplikácia pozostáva z dvoch komponentov. Serverom je Node.js aplikácia, ktorá pracuje so skupinou skriptov vygenerovaných pomocou IDE a databázou MongoDB. IDE umožňuje nasadenie skriptov na platformách Windows, OSX, Linux, Raspberry Pi a ARM-6+. Ich vytvorenie prebieha v troch krokoch. V prvom kroku sa pridajú želané zariadenia. Jedno zariadenie je definované pomocou IP adresy, komunikačného protokolu, portu a ďalších informácií podľa potreby. Podporované sú rozličné protokoly, napríklad TCP, UDP, Serial, SSH, HTTP, MQTT a ďalšie. Druhý krok spočíva v pridaní príkazov pre jednotlivé zariadenia. Napokon sa jednoduchým ťahaním vytvorených príkazov vytvorí dizajn ovládača.

3.2 Domoticz

Serverová časť systému Domoticz je naprogramovaná v jazyku C++. Vyžaduje zložitejšiu konfiguráciu. Medzi systémové požiadavky patrí okrem iného 256MB

pamäte a 200MB miesta na disku. Ponúka automatickú detekciu zariadení pripojených cez USB port, pripojenie zariadení v lokálnej sieti, zdieľanie zariadení s priateľmi či pripojenie vzdialeného servera. Ďalšími z mnohých možností sú emailové alebo push notifikácie a logovanie histórie. Automatické zapínanie a vypínanie svetla na základe východu a západu slnka je možné nastaviť zadaním zemepisnej šírky a dĺžky, ktoré si vie používateľ nechať automaticky vyhľadať pomocou tejto aplikácie. Samozrejmosťou je aj identifikácia používateľa pred prístupom do systému. Používateľské prostredie v HTML5 je škálovateľné a prispôsobivé klasickým webovým prehliadačom aj ich mobilným verziám.

3.3 Home Assistant

Ďalším automatizačným systémom, na ktorý sme sa pozreli, je Home Assistant. Back-end je naprogramovaný v jazyku Python, štandardne využíva súborovo-orientovaný databázový systém SQLite, no umožňuje aj používanie databázového servera. Pozostáva z viacerých častí. Jadro tvorí komunikačná zbernica, ktorá čaká a reaguje na udalosti komponentov a časovača, volania služieb alebo zmenu ich stavu. Jadro komunikuje so samotnými senzormi a aktuátormi, používateľovými príkazmi a automatizačnými pravidlami. Na ovládanie systému Home Assistant používateľ potrebuje webový prehliadač. Tiež poskytuje širokú škálu funkcionality, ako je napríklad ovládanie svetla, kúrenia či elektrospotrebičov, ale aj lokalizáciu osôb či detekciu prítomnosti v priestoroch domu.

3.4 Home.Pi

Už z názvu Home.Pi je zrejmé, že toto automatizačné riešenie je primárne určené pre Raspberry Pi. Autor riešenia prirovnáva jeho architektúru k mikroslužbám. Jednotlivé komponenty je možné vymieňať bez narušenia celého systému. Väčšina komponentov beží na cloude. Veľmi jednoduché používateľské rozhranie určené pre mobilné telefóny je vytvorené pomocou Ionic Creator. V porovnaní s predchádzajúcimi riešeniami tento menší projekt neposkytuje širokú škálu funkcionality a je zjavné, že autor v jeho vývoji už nepokračuje.

3.5 Node-RED

Node-RED je nástroj na prepájanie hardvérových zariadení, online služieb a rozhraní. Editor spustený vo webovom prehliadači umožňuje vytváranie grafov prepojení rôznych uzlov z palety. Používateľ má možnosť vytvárať grafy toku informácií a akcií, funkcie v JavaScripte a šablóny, ktoré môže následne uložiť a znovu použiť. Node-RED je postavený na Node.js, vďaka čomu je možné nasadiť ho ako na cloude, tak aj na nízkonákladovom hardvéri akým je Raspberry Pi. Jeho primárnym cieľom je spracovanie dát pred ich odoslaním.

3.6 openHAB

Integračné riešenie openHAB alebo open Home Automation Bus je platforma pre Smart Home. Distribuovaná architektúra SOA(*Service Oriented Architecture*) umožňuje prepojenie s ďalšími automatizačnými systémami, zariadeniami a technológiami do jedného riešenia. Sprostredkuje jednotné používateľské rozhranie a spoločný prístup pre celý systém. Je prezentovaný ako najflexibilnejší domáci automatizačný systém, ktorý umožní používateľovi splniť akékoľvek predstavy o inteligentnom domove. Jeho nasadenie však už vyžaduje veľa času a trpezlivosti. K dispozícii sú podrobné príručky a postupy. Napriek tomu, že je možné openHAB nasadiť na Raspberry Pi, odporúča sa využiť ho nanajvýš pri experimentovaní. Limity Raspberry Pi môžu viesť k nestabilite a slabému výkonu. Na vývoji openHAB sa ešte stále pracuje.

3.7 Sumarizácia

Väčšina riešení je monolitického charakteru alebo využíva distribuovanú architektúru SOA, ktorá umožňuje prepojenie viacerých automatizačných modulov, no ich jadro je opäť komplexná monolitická aplikácia.

My sa budeme zaoberať tým, ako takéto monolitické riešenie rozbiť na menšie časti - akési mikroslužby. V nasledujúcej časti popíšeme architektúru mikroslužieb a vlastnosti, ktoré ju odlišujú od iných architektúr.

4 Výber architektúry

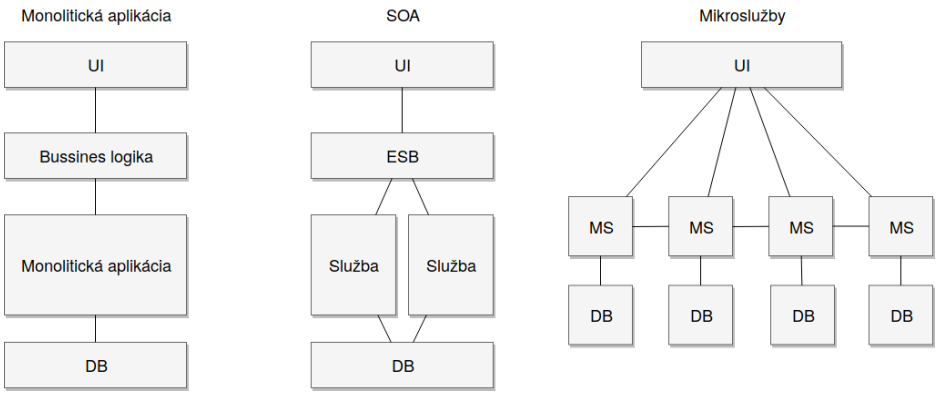
V tejto sekcii popíšeme a odôvodníme výber architektúry mikroslužieb. Jeden zo spôsobov rozdelenia architektúr je na monolitické a distribuované. Medzi distribuované architektúry patria už spomínané mikroslužby a SOA. Voľba distribuovanej architektúry je jednoznačná v prípade integračného softvéru pre domáce automatizačné riešenie. Kvôli rýchlemu vývoju oblasti internetu vecí je náročné plánovať a odhadnúť, ako sa vyvinie celý proces tvorby komplexného softvéru. Je potrebná zložitá integrácia zariadení, dát a aplikácií.

Softvérová architektúra nazývaná mikroslužby je čoraz využívanejšou a žiadanejšou voľbou pri vývoji softvéru. Podľa článku [7] pod pojmom mikroslužby rozumieme prístup vývoja aplikácie ako skupiny malých služieb, z ktorých každá beží ako samostatný proces komunikujúci odľahčenými mechanizmami, často pomocou HTTP zdrojov dát prístupných cez nejaké API. Tieto služby sú vyvíjané samostatne a nezávisle od ostatných. Existuje len minimum centralizovaného manažmentu týchto služieb. Služby môžu byť implementované v rôznych programovacích jazykoch, môžu používať ľubovoľné technológie, úložiska dát a môžu bežať na rôznych platformách. Ďalšou charakteristickou črtou popísanou v [8] je, že jedna mikroslužba je zodpovedná za jedinú úlohu. To znamená, že vykonáva jedinú prácu, ktorá je ľahko popísateľná.

Základnou črtou architektúry SOA, rovnako ako pri mikroslužbách, je rozdelenie aplikácie na menšie, potenciálne distribuované služby. Tieto architektúry sa líšia v niekoľkých vlastnostiach:

- **rozdelenie na menšie časti**
  - mikroslužby delia aplikáciu na *malé* služby vykonávajúce jedinú úlohu,
  - SOA delí aplikáciu na *rôzne veľké* časti, ktoré môžu byť aj komplexné,
- **zdieľanie komponentov**
  - mikroslužby sa snažia držať zásady *„share-as-little-as-possible“*, čiže zdieľajú medzi sebou len to najnutnejšie,
  - služby v SOA medzi sebou často zdieľajú rôzne komponenty, napríklad databázu,
- **komunikácia**
  - SOA zvyčajne využíva množstvo rôznych komunikačných protokolov na komunikačnej zbernici (ESB - *Enterprise Service Bus*), ktorá prepája jednotlivé služby a v konečnom dôsledku sa stáva problémom pri potrebe škálovania,
  - mikroslužby komunikujú medzi sebou jednotne prostredníctvom API vrstvy alebo zasielaním správ pomocou jednoduchého protokolu.

Obrázok 1 znázorňuje základné rozdiely medzi architektúrami monolitickej aplikácie, SOA a mikroslužbami.



Obr. 1: Znázornenie rozdielov medzi architektúrou monolitickej aplikácie, SOA a mikroslužbami.

Mikroslužby využívajú aj obrovské spoločnosti ako Amazon či Netflix, čo im v konečnom dôsledku dovolilo rásť do takej veľkosti, akú do dnešného dňa dosiahli.

S mikroslužbami prichádza mnoho výhod:

- **jednoduchosť**
  - vývoj a testovanie mikroslužby sú rýchle a nezávislé v porovnaní s vývojom komplexnej monolitickej aplikácie,

  - na každej mikroslužbe môže pracovať iný tím vývojárov, ich práca sa navzájom neovplyvňuje,
- **škálovateľosť**
  - jednotlivé mikroslužby môžu byť podľa potreby škálované bez zásahu do ostatných častí systému,
- **nezávislosť**
  - každá mikroslužba beží ako samostatný proces,
  - možnosť zvoliť pre každú službu najvhodnejšie technológie,
- **robustnosť**
  - odolnosť voči chybám,
  - znižuje sa riziko výpadku systému, pretože chyba jednej služby nemusí ovplyvniť chod ostatných častí systému,
- **flexibilita**
  - možnosť pripojenia rôznorodých zariadení do siete,
- **optimalizácia**
  - využitia výpočtovej sily a systémových prostriedkov.

Takisto prinášajú aj mnoho výziev, s ktorými sa treba vysporiadať:

- **efektívne rozdelenie**
  - zvážiť prínos použitia architektúry mikroslužieb,
  - analyzovať spôsob rozdelenia funkcionality na menšie časti,
- **bezpečnosť**
  - riziká spojené s komunikáciou mikroslužieb po sieti,
- **koordinácia**
  - zabezpečiť, aby sa systém stále navonok správal ako jedna aplikácia,
  - efektívna spolupráca mikroslužieb,
  - vysporiadať sa s oneskorenými, nedoručenými alebo duplikovanými správami,
- **konfigurácie a manažment**
  - zvýšený počet konfigurácií zapríčinený zvýšeným počtom komponentov,
  - automatizácia testovania a nasadenia.

Mikroslužby spĺňajú mnoho z vlastností popísaných v kapitole 2, preto má zmysel zaoberať sa ich použitím pri návrhu integračného riešenia. Typické funkcie integračnej aplikácie preberie niekoľko nezávislých softvérových komponentov. V tejto práci analyzujeme, ako rozdeliť úlohy integračného riešenia. Jedna z možných kategorizácií je podľa zamerania na riadenie prístupu, správu dát, správu zariadení, spracovanie udalostí, externú integráciu, monitoring a iné. Ďalšiu analýzu si vyžaduje návrh komunikácie medzi mikroslužbami. Na bezpečnosť komunikácie sa môžme pozerať dvojako. Okrem zabezpečenia voči vonkajším vplyvom je potrebné zaistiť, aby spolu dokázali komunikovať len tie mikroslužby, ktorých komunikácia je nevyhnutná.

5 Komunikácia medzi mikroslužbami

Komunikácia medzi jednotlivými komponentami patrí k najdôležitejším častiam návrhu aplikácie, ktorá využíva architektúru mikroslužby. V monolitickej aplikácii, ktorá beží ako jeden proces, komponenty volajú metódy a funkcie iných komponentov na úrovni programovacieho jazyka. Na druhej strane mikroslužby, ktoré bežia ako samostatné procesy, potrebujú nejaký komunikačný protokol na zabezpečenie komunikácie a kooperácie. V tejto sekcii popíšeme niekoľko prístupov, ktoré sú často využívané na komunikáciu a prenos dát, ich výhody a nevýhody.

5.1 HTTP komunikácia

HTTP [13] (*Hypertext transfer protocol*) je protokol aplikačnej vrstvy, ktorý využíva klient-server architektúru. Je najviac využívaným protokolom pri komunikácii webových služieb. Spomenieme niekoľko možností využívajúcich HTTP.

5.1.1 SOAP [10] (*Simple Object Access Protocol*) je protokol založený na XML, ktorý umožňuje výmenu dát medzi aplikáciami. Zvyčajne využíva aplikačný protokol HTTP, ale podporuje aj iné protokoly. Je navrhnutý tak, aby ho bolo možné využiť v kombinácii s ľubovoľným programovacím jazykom na ľubovoľnej platforme. Keďže samotný XML sa radí medzi odľahčené jazyky na výmenu dát, aj SOAP patrí medzi „ľahké“ protokoly. I keď stále má svoje uplatnenie najmä vo väčších projektoch, stáva sa čoraz menej populárnym. Jedným z dôvodov je aj to, že je príliš komplikovaný. Jednoduchá požiadavka môže vyzerať napríklad takto:

```
POST /Students HTTP/1.1
Host: www.example.sk
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:m="www.example.sk">
  <soap:Header></soap:Header>
  <soap:Body>
    <m:GetStudentsAge>
      <m:StudentsName>Thomas</m:StudentsName>
    </m:GetStudentsAge>
  </soap:Body>
```

5.1.2 REST [11] (*Representational state transfer*) je štandardom dnešnej doby. Podobne ako SOAP, aj REST má podporu pre použitie s rôznymi protokolmi aplikačnej vrstvy, no vo väčšine prípadov využíva protokol HTTP. Pre zasielané dáta najčastejšie používa formát JSON. Je jednoduchý, práca s ním nevyžaduje žiadne veľké schopnosti. Je podporovaný obrovským množstvom knižníc v rôznych programovacích jazykoch ako pre klienta, tak aj pre server. Príklad požiadavky:

```
GET http://example.sk/students/age/Thomas
```

Odpoveď na predchádzajúcu požiadavku:

```
{
  "age": "42"
}
```

5.1.3 GraphQL [12] je pomerne nový dopytovací jazyk. Narozdiel od REST a SOAP dáva klientovi možnosť vypýtať si presne také dáta, aké potrebuje, namiesto presného definovania možných požiadavok a štruktúry dát zahrnutých v odpovedi na strane servera. Tým zabraňuje v prenose zbytočného množstva nepotrebných dát. Príklad požiadavky a odpovede je znázornený na obrázku **??**. Požiadavka môže vyzerať napríklad takto:

```
{
  student(id: "15"){
    name
    age
  }
}
```

Odpoveď na predchádzajúcu požiadavku:

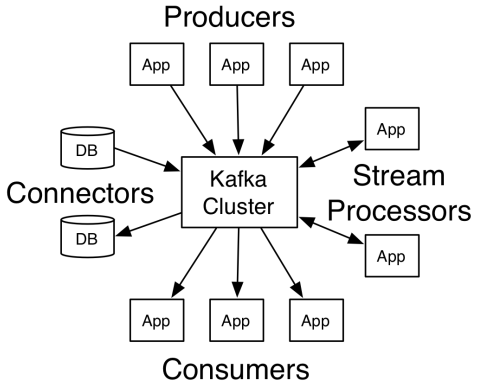
```
{
  "data": {
    "student": {
      "name": "Thomas Angelo",
      "age": 42
    }
  }
}
```

5.2 Komunikácia riadená správami

Ďalšou možnosťou je komunikácia posielaním správ. Narozdiel od vyššie spomínanej komunikácie prostredníctvom protokolu HTTP, ktorá je synchrónna, komunikácia posielaním správ prebieha asynchrónne. Oba spôsoby majú za cieľ výmenu dát medzi koncovými zariadeniami v sieti. V predchádzajúcich prípadoch

klient posiela požiadavky na server, ktorý je poskytovateľom služieb. Rozhranie servera ponúka služby identifikované pomocou URI identifikátora (REST), či tela XML požiadavky (SOAP). Pri komunikácii posielaním správ prístupné služby môžu byť identifikované jednotlivými schránkami, do ktorých sú správy posielané.

5.2.1 Apache Kafka je distribuovaná streamovacia platforma, ktorá sa využíva na prijímanie a spracovanie prúdov dát v reálnom čase. Využíva vlastný aplikačný protokol. Kombinuje dva modely, a to *publish-subscribe* model a model radov. Prúdy dát sú generované množstvom dátových zdrojov, ktoré ich posielajú simultánne. Umožňuje horizontálne škálovanie a replikáciu správ na viacero serverov pre prípad, že niektorý z nich zlyhá. Apache Kafka ponúka tri hlavné funkcie - posielanie správ a ich odber z jednotlivých tokov dát, efektívne ukladanie dát v poradí, v akom boli vygenerované a spracovanie týchto tokov dát v reálnom čase. Obrázok 2 zobrazuje schému, ktorá zahŕňa Kafka klaster, producentov, ktorí generujú dáta, konzumentov, ktorí tieto dáta odoberajú, procesory spracúvajúce toky dát a konektory, ktoré môžu napríklad ukladať dáta do sekundárneho úložiska pre ich ďalšie spracovanie a analýzu. Apache Kafka sa využíva viac v systémoch spracujúcich *big data*.



Obr. 2: Apache Kafka[1].

5.2.2 MQTT (*Message Queuing Telemetry Transport*) je rovnako ako HTTP protokol aplikačnej vrstvy, ktorý využíva *publish-subscribe* model. Jeho cieľom

[1] Zdroj: `https://kafka.apache.org/23/images/kafka-apis.png`

je spoľahlivé doručenie v nespoľahlivej sieti alebo v sieti s obmedzenými prostriedkami. Je jednoduchý a nenáročný, preto je využívaný v oblasti internetu vecí, kde je často problém s obmedzenými zdrojmi.

5.3 Zhrnutie

Hoci sú SOAP a REST veľmi rozšírené v distribuovanom prostredí, ich použitie je vhodnejšie v prípade klient-server architektúry. Jedným zo základných predpokladov pre mikroslužby je, že sú nezávislé a voľne spojené. To sa pri použití synchrónnej komunikácie nedá docieliť, napriek tomu sa však REST často používa aj na komunikáciu mikroslužieb, ak je pre mikroslužbu nevyhnutné, aby dostala odpoveď synchrónne. Je ale zjavné, že komunikácia založená na odosielaní správ, akú ponúka protokol MQTT, nám poskytuje väčšiu nezávislosť a lepšie možnosti škálovania.

V oblasti internetu vecí má protokol MQTT viacero výhod. Podstatným rozdielom je, že pri komunikácii prostredníctvom HTTP protokolu klient musí poznať IP adresu servera, ktorého služby chce využiť. Na komunikáciu je potrebné, aby klient a server boli súčasne pripojení. Na druhej strane komunikácia MQTT protokolom je asynchrónna, teda na komunikáciu sa nevyžaduje, aby boli príjemca a odosielateľ správy súčasne pripojení, nepotrebujú poznať adresu toho druhého, ani mať žiadnu vedomosť o jeho existencii. Spôsob komunikácie protokolom HTTP nie je vhodný v prípade, kedy nepoznáme adresu zariadenia, s ktorým chceme komunikovať. V oblasti internetu vecí, kde IoT zariadenia sú umiestnené v sieti s routerom, ktorý používa NAT, nie je priama komunikácia ideálna. Pri náraste počtu IoT zariadení je nepredstaviteľné, aby malo každé z nich priradenú verejnú IP adresu. Je však žiadané, aby zariadenia komunikovali s inými zariadeniami a službami aj mimo lokálnej siete. Navyše architektúra IoT systému je zvyčajne dynamická, jednotlivé IoT zariadenia a ich adresy sa môžu v priebehu času meniť. Pri použití publish-subscribe modelu stačí, aby sa vedeli všetky zariadenia pripojiť na MQTT broker.

Treba zdôrazniť aj fakt, že komunikácia prostredníctvom MQTT protokolu vyžaduje omnoho nižšiu spotrebu energie v porovnaní s protokolom HTTP, čo je obrovskou výhodou, keďže IoT zariadenia sú zvyčajne napájané batériou. Nižšia spotreba energie vyplýva aj z asynchrónnej komunikácie. IoT zariadenia môžu byť aktívne v určených časových intervaloch, počas ktorých prijímajú a odosielajú správy alebo sa prebrať z neaktívneho stavu pri vzniku nejakej udalosti.

Z vyššie uvedených dôvodov sme sa pri návrhu integračného riešenia pre Smart Home rozhodli využiť protokol MQTT.

6 Protokol MQTT

V tejto sekcii bližšie popíšeme protokol MQTT, jeho výhody a nevýhody. MQTT je aplikačný protokol, ktorý využíva TCP protokol transportnej vrstvy.

6.1 Model publish-subscribe

V porovnaní s bežným klient-server modelom, kde klient predpokladá existenciu bežiaceho servera, na ktorý sa pripojí, v publish-subscribe modeli sa publisher aj subscriber správajú ako klienti, ktorí na to, aby spolu komunikovali, nepotrebujú vedieť o lokalite ani existencii toho druhého. Klienti komunikujú tak, že publisher posiela správy na centrálneho agenta, ktorý sa nazýva broker. Cez neho prebieha všetká komunikácia, klienti medzi sebou nikdy nekomunikujú priamo.

Každý klient sa najskôr musí pripojiť pomocou správy CONNECT, na čo následne broker odpovedá správou CONNACK, ktorá informuje klienta, či bolo pripojenie úspešné. Správa CONNECT obsahuje niekoľko povinných a nepovinných informácií. Medzi povinné patria:

- **clientId** - unikátny identifikátor klienta, ktorý broker využíva pri ukladaní aktuálneho stavu klienta,
- **cleanSession** - ak je tento parameter nastavený na *true*, broker si nebude o klientovi ukladať žiadne informácie, vrátane topicov, na ktorých odber sa prihlásil.

Ďalšie nepovinné parametre:

- **username** a **password** - sa využívajú pri autentifikácii klientov,
- **lastWillTopic**, **lastWillQos**, **lastWillMessage** a **lastWillRetain** - popisujú poslednú správu, ktorá sa odošle pred neočakávaným odpojením klienta,
- **keepAlive** - je čas v sekundách, ktorý určuje, ako dlho má broker udržiavať spojenie s klientom, ak medzi nimi neprebieha odosielanie správ.

Po pripojení sa už klient môže na brokeri prihlásiť(subscribe) na odber správ z takzvaných topicov alebo do nich odosielať(publish) správy.

6.2 Topic

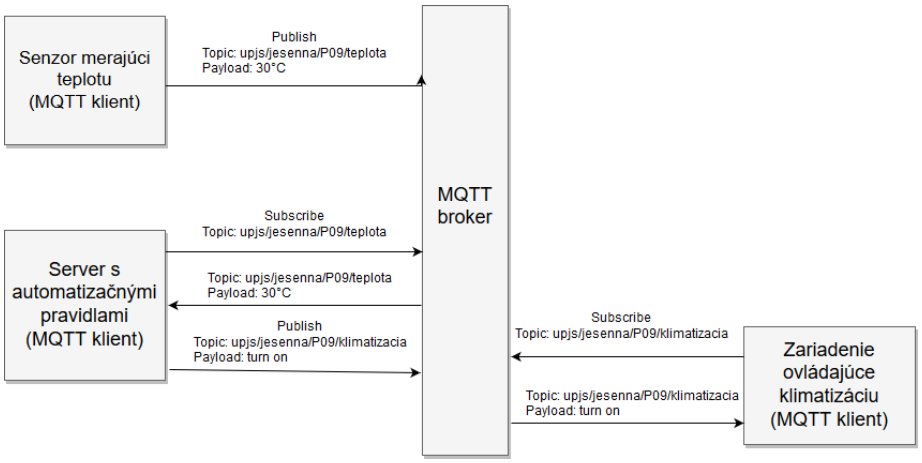
Topicom je hierarchická štruktúra, ktorá môže pozostávať z viacerých úrovní. Tie sú v názve topicu oddelené lomkou. Príkladom topicu môže byť napríklad *upjs/jesenna/p09/teplota*, pričom prihlásením na tento topic by sme dostávali správy o teplote v posluchárni P09 v budove našej univerzity na Jesennej ulici.

Klient má možnosť prihlásenia na konkrétny topic, ako je uvedené v príklade, alebo sa môže prihlásiť na odber z celej skupiny topicov použitím špeciálnych znakov. Príkladom môže byť *upjs/jesenna/+/teplota*. Špeciálny znak "+"nahrádza ľubovoľný popis jednej úrovne v hierarchii. V prípade nášho príkladu by klient prihlásený na takýto topic dostával správy o teplote z každej miestnosti v budove na Jesennej ulici. Ďalším špeciálnym znakom je "#". Používa sa výhradne na konci topicu a môže nahradiť viac úrovní. Napríklad, po prihlásení na topic *upjs/jesenna/#* bude klient dostávať správy odoslané do všetkých topicov, začínajúcich na *upjs/jesenna/*, teda všetky dáta v rámci budovy na Jesennej ulici.

Keď publisher odošle do nejakého topicu správu, tú príjme každý subscriber, ktorý je na danom topicu prihlásený. Správa PUBLISH obsahuje nasledujúce informácie:

– **packetId** - unikátny identifikátor paketu,
– **topicName** - názov topicu,
– **qos** - *Quality of Service* definuje garanciu doručenia správy, môže mať tri hodnoty:
  • **0** - najviac raz,
  • **1** - aspoň raz,
  • **2** - práve raz,
– **retainFlag** - ak je hodnota nastavená na *true*, broker uloží správu a nový klient po pripojení na topic túto správu dostane,
– **payload** - obsah správy,
– **dupFlag** - indikuje, či je to správa poslaná opakovane po tom, čo neobdržala potvrdenie o doručení pôvodnej správy.

Každý klient v tomto modeli môže byť súčasne subscriberom aj publisherom. Môže ním byť ľubovoľné zariadenie ako senzor, či aktuátor, server(back-end) aj klient(front-end). Dvaja klienti môžu komunikovať asynchrónne, teda nepotrebujú bežať v rovnakom čase na to, aby komunikácia úspešne prebehla. V prípade, že je klient odpojený, všetky správy z topicov, na ktoré je prihlásený, mu po opätovnom prihlásení na broker budú doručené. Obrázok 3 znázorňuje príklad komunikácie pomocou protokolu MQTT.



Obr. 3: Príklad komunikácie protokolom MQTT

Značnou výhodou protokolu MQTT je obrovské množstvo implementácií brokerov, rovnako ako aj klientov, či už v podobe desktopových aplikácií, alebo knižníc pre rôzne programovacie jazyky. Spomenieme si niekoľko z nich v nasledujúcich sekciách.

7 MQTT broker

MQTT broker je vyššie spomínaný centrálny agent, ktorý prijíma správy od pripojených klientov, filtruje ich a rozposiela klientom, ktorí sú prihlásení na odber z príslušného topicu. V tejto sekcii predstavíme niekoľko brokerov a popíšeme ich vlastnosti a možnosti. Zdôvodnime výber konkrétneho brokera.

7.1 HiveMQ

Jedným z MQTT brokerov je HiveMQ, implementovaný v jazyku Java. Má podporu pre Linux, Windows, OS X, ale môže byť spustený aj na cloude. Medzi systémové požiadavky patrí minimálne 4GB pamäte RAM, aspoň 4 CPU, 10GB voľného miesta na disku či OpenJDK JRE v minimálnej verzii 11. Je vhodnejší skôr pre klasické servery.

Na výber je z troch verzií, a to open-source HiveMQ *Community Edition* alebo *Professional* či *Enterprise Edition*, na použitie ktorých je potrebná licencia. Viacvláknový prístup umožňuje až 10 miliónov pripojených zariadení súčasne s minimálnym zdržaním. Implementuje každý z levelov QoS. Má podporu pre radenie správ do fronty v prípade nedostupnosti klienta i podporu pre distribuovanú architektúru klastrov, čím zamedzuje problému jediného bodu zlyhania, straty dát a neprístupnosti brokera. Snaží sa o maximálnu podporu škálovateľnosti, ktorú však zabezpečuje predovšetkým v platených verziách. Samozrejmosťou je šifrovanie a ďalšie formy zabezpečenia. Nasledujúca tabuľka 1 popisuje niekoľko rozdielov v podpore medzi jednotlivými edíciami. Je zrejmé, že ak používateľ chce čosi viac, než len základnú funkcionalitu, potrebuje si zaobstarať licencovanú verziu.

	Community	Professional	Enterprise
Websockets	✓	✓	✓
IPv4 & IPv6	✓	✓	✓
TLS / SSL	✓	✓	✓
Linear Scaling Shared Subscription Sharding	×	✓	✓
Linux Epoll Support	×	✓	✓
Proxy Protocol	×	×	✓
Cluster Support	×	✓	✓
Real-time monitoring Dashboard	×	✓	✓
MQTT Client Drill-Down Analysis	×	✓	✓
Advanced Analysis	×	×	✓
TLS / SSL for MQTT	✓	✓	✓
Pluggable Authentication	✓	✓	✓
Authorization and Permissions	✓	✓	✓

Tabuľka 1: Porovnanie funkcionality verzií HiveMQ brokerov.

7.2 Eclipse Mosquitto

Ďalším brokerom je Eclipse Mosquitto [5]. Je open-source a vďaka svojej nenáročnosti je ideálny nielen pre bežné servery ale aj pre zariadenia s obmedzenými výpočtovými zdrojmi. Je vhodný pre širokú škálu platforiem. Je implementovaný v jazykoch C a C++. Jeho inštalácia je veľmi rýchla a jednoduchá. Broker je možné nakonfigurovať pomocou konfiguračného súboru *mosquitto.conf*.

7.2.1 Konfigurácia Na použitie brokera nie je konfigurácia nutná, broker môže bežať so štandardnými nastaveniami. Popíšeme si niektoré užitočné možnosti:

- **Autentifikácia:** Pri použití predvolených nastavení nie je potrebné, aby sa klient autentifikoval pred tým, než začne s posielaním správ alebo prihlásením na odber z topicov. Samotný MQTT protokol ponúka možnosť autentifikácie pomocou prihlasovacieho mena a hesla. Tie sa definujú prostredníctvom možnosti *password_file*. Je možné ich nastaviť globálne alebo osobitne pre každý port, na ktorom bude broker počúvať. Ďalšou možnosťou je použitie certifikátu na zabezpečenie šifrovania alebo zdieľaného kľúča. Spôsoby autentifikácie je možné ľubovoľne kombinovať.
- **Listener:** Predvolený listener počúva na porte 1883. Tento port je možné zmeniť alebo vytvoriť ďalšie listenery, ktoré budú počúvať na iných portoch. Pre listener máme možnosť ďalších nastavení. Vieme nakonfigurovať, aby počúval iba na špecifikovanej IP adrese. Listener môže používať predvolený protokol MQTT alebo protokol websocket. Websocket[9] poskytuje obojsmernú komunikáciu prostredníctvom jedného TCP spojenia. Je vhodný na komunikáciu s webovým prehliadačom.
  Ponúka aj možnosť obmedzení pre skupinu klientov tým, že pre nich definujeme prefix, ktorým určíme, ku ktorej časti hierarchie bude mať klient prístup.
- **Bridge:** Broker má možnosť byť nakonfigurovaný tak, aby fungoval ako bridge. Takým spôsobom môžeme prepojiť navzájom dva brokery. Zvyčajne sa používajú na zdieľanie správ medzi systémami. Je možné definovať, z ktorých topicov budú správy preposielané medzi jednotlivými dvojicami brokerov, pričom môžeme modifikovať topic, do ktorého sa správy prepošlú.
- **Ďalšie možnosti:**
  - **acl_file -** Súbor, v ktorom vieme definovať povolenia prístupu pre jednotlivých klientov. Práva k čítaniu alebo posielaniu správ do topicov sú určované pomocou kľúčových slov *read*, *write* alebo *readwrite*. Práva môžu byť definované aj pre skupinu topicov s využitím špeciálnych znakov + a #, alebo aj s využitím substitúcie nasledovne: %c sa nahradí identifikátorom klienta a %u jeho používateľským menom.
  - **auth_plugin -** Špecifikuje cestu k externému modulu pre autentifikáciu a riadenie prístupu. Môže byť použitých aj viac modulov.
  - **max_queued_messages -** Definuje maximálny počet pre správy zaradené do fronty pre jedného klienta. Predvolená hodnota je 100. Hodnotou 0 sa definuje neobmedzený počet správ, ale neodporúča sa toto nastavenie používať.

- **persistence -** Ak je nastavené na *true*, informácie o pripojeniach, prihláseniach na odber z topicov a správach v nich budú uložené na disku a pri reštarte brokera budú odtiaľ načítané.

Vyššie spomínané možnosti sú len veľmi malou časťou veľkého množstva možných nastavení, ktoré Mosquitto broker ponúka.

7.3 Moquette

Broker Moquette je implementovaný v jazyku Java. Konfiguračný súbor má rovnaký formát ako Eclipse Mosquitto broker. Môže bežať samostatne alebo ako súčasť iného projektu. Je nenáročný a jeho inštalácia je jednoduchá.

7.4 Mosca

Mosca je Node.js broker, ktorý rovnako ako Moquette môže byť použitý samostatne, ale aj ako súčasť iného projektu. Narozdiel od vyššie spomínaných brokerov má podporu iba pre správy s QoS 0 a 1. Čo sa týka využitia a konfigurácie, Mosca ponúka oproti iným spomínaným brokerom iba základné možnosti. Perzistenciu poskytuje s využitím databáz Level, MongoDB alebo Redis.

7.5 VerneMQ

VerneMQ je výkonný distribuovaný MQTT broker. Efektívne využíva všetky dostupné prostriedky pre jednoduché vertikálne škálovanie. Keďže je distribuovaný, zaručuje odolnosť voči zlyhaniu a ponúka aj horizontálne škálovanie. Má podporu pre klastrovanie, brige, autentifikáciu a autorizáciu pomocou databáz PostgreSQL, MySQL, Redis a MongoDB, šifrovanie a mnoho ďalších funkcií.

7.6 Zhrnutie

Mnoho brokerov ponúka podobné možnosti konfigurácie, autentifikáciu, bridgovanie, nastavenie listenerov. Niektoré, ako napríklad Mosca, ponúkajú len nejaký ich základný výber. Líšia sa v implementácii a cieľovom použití. Niektoré sú vhodnejšie pre menšie projekty, iné majú schopnosti zvládnuť aj milióny pripojených klientov.

Okrem MQTT brokerov existujú aj iné komunikačné servery, ktoré majú podporu pre protokol MQTT. Príkladom je Apache ActiveMQ, open-source server na posielanie správ, postavený na Jave. Podporuje komunikáciu použitím protokolu MQTT, ale aj AMQP a STOMP. Prístupné sú aj možnosti použitia brokerov na cloude. CloudMQTT ponúka prístup na Mosquitto broker v rôznych cenových hladinách v závislosti od požiadaviek na prenosovú rýchlosť, počet pripojení a podobne.

Pri našom návrhu integračného riešenia pre „Smart Home“ sme sa rozhodli využiť Eclipse Mosquitto broker. Je open-source, ponúka veľa možností

prispôsobenia a je jednoduchý na inštaláciu a použitie, vďaka čomu je vhodný na účely analýzy, vývoja a testovania.

V ďalšej sekcii budeme popisovať niektoré časti konfigurácie MQTT brokera. Popíšeme, ako je možné nahradiť point-to-point komunikáciu medzi jednotlivými komponentami systému.

8 MQTT klient

Ako sme už vyššie spomenuli, existuje množstvo knižníc a implementácií MQTT klientov. Máme k dispozícii nástroje pre prácu s MQTT klientom v rôznych programovacích jazykoch, vďaka čomu nie sme obmedzení pri výbere najvhodnejšieho jazyka a technológií pre každý komponent. Pri implementácii jednotlivých komponentov systému preto využívame viacero z nich. V tejto sekcii si ich niekoľko spomenieme.

8.1 Eclipse Paho

Projekt Eclipse Paho [14] poskytuje implementácie štandardných protokolov na posielanie správ. Obsahuje implementácie MQTT klienta pre: Java, Python, JavaScript, GoLang, C, C++, Rust, .Net(C#), Android Service a Embedded C/C++. Nasledujúca tabuľka 2 zobrazuje rozdiely v podpore niektorých z týchto knižníc.

	Java	Python	JavaScript	C	C++	Android Service	Embedded C/C++
MQTT 3.1	✓	✓	✓	✓	✓	✓	✓
MQTT 3.1.1	✓	✓	✓	✓	✓	✓	✓
MQTT 5.0	×	×	×	✓	×	×	×
LWT	✓	✓	✓	✓	✓	✓	✓
SSL / TLS	✓	✓	✓	✓	✓	✓	✓
Automatic Reconnect	✓	✓	✓	✓	✓	✓	×
Offline Buffering	✓	✓	✓	✓	✓	✓	×
Message Persistance	✓	✓	×	✓	✓	✓	×
WebSocket Support	✓	✓	✓	✓	×	✓	×
Standard MQTT TCP Support	✓	✓	×	✓	✓	✓	✓

Tabuľka 2: Porovnanie Eclipse Paho MQTT klientov.

V tabuľke je vidieť, že každá má podporu pre nastavenie LWT (Last Will and Testament messages), čo znamená, že vieme nastaviť, čo sa udeje a aká správa sa odošle pred náhlym odpojením klienta. Všetky majú podporu pre TLS/SSL.

Väčšina z nich podporuje ukladanie správ do zásobníka počas doby, keď klient nie je pripojený a tie sa pri opätovnom pripojení odošlú. Tiež majú podporu pre automatické pripojenie pri strate spojenia. Tieto dve činnosti nepodporuje knižnica pre mikrokontroléry (Embedded C/C++) ani knižnica pre .Net.

Štandardnú podporu pre komunikáciu s MQTT brokerom pomocou TCP spojenia má každá knižnica s výnimkou JavaScript knižnice, ktorá má výhradne podporu pre komunikáciu protokolom WebSocket, ktorý slúži pre komunikáciu webového prehliadača s MQTT brokerom. Túto knižnicu využívame pri implementácii komponentu pre vizualizáciu dát a ovládanie systému.

8.2 Mosquitto

Knižnica implementovaná v jazyku C je vyvíjaná rovnako ako Paho spoločnosťou Eclipse Foundation. Má podporu pre QoS 0,1 aj 2. Mosquitto-PHP je rozšírením tejto knižnice, ktoré umožňuje jej použitie v kombinácii s PHP. Majú podporu pre LWT, TLS a SSL.

8.3 ruby-mqtt

Knižnica implementuje MQTT protokol pre jazyk Ruby. Nemá podporu pre automatické znovu-pripojenie ani podporu pre QoS 2.

8.4 MQTT.js

Knižnica MQTT.js napísaná v jazyku JavaScript pre platformu Node.js a webové prehliadače. Umožňuje komunikáciu protokolom WebSocket aj MQTT. Podporuje QoS 0,1 aj 2.

8.5 HiveMQ

Java knižnica, ktorá je kompatibilná s verziami MQTT 3.1.1 aj 5.0 poskytuje vysoký výkon. Podporuje všetky QoS, SSL/TLS, štandardnú komunikáciu cez TCP aj WebSockets.

8.6 Ďalšie knižnice a nástroje

Okrem vyššie spomínaných existujú ďalšie knižnice, ako napríklad MQTT-C pre mikrokontroléry, CocoaMQTT pre iOS a OS X napísana v jazyku Swift, gmqtt a hbmqtt pre jazyk Python a mnoho ďalších. Medzi štandardne poskytovanú funkcionalitu patrí autorizácia, šifrovanie či nastavenie „poslednej vôle“ pre prípad nečakaného odpojenia klienta, samozrejme možnosti odoslania správ a prihlásenie na odber správ z topicov, a tiež možnosť implementovať udalosť, ktorá vznikne ako reakcia na prichádzajúcu správu. Voľba knižnice zvyčajne závisí najmä od použitého zariadenia a programovacieho jazyka.

Okrem knižníc implementujúcich MQTT klienta sú k dispozícii aj užitočné nástroje, fungujúce ako MQTT klient s používateľským prostredím na monitorovanie činnosti v topicoch. Jedným z nich je napríklad **mqtt-spy** alebo jeho verzia pre konzoly **mqtt-spy-daemon**. Nasledujúca tabuľka 2 zobrazuje ich funkcie. Nástroj mqtt-spy sme využívali v priebehu vývoja prototypu integračného

	mqtt-spy-deamon	mqtt-spy
Používateľské prostredie	Konzola	GUI
Požadovaná verzia jazyka Java	Java 7+	Java 8 akt. 60+
Podporovaná verzia MQTT protokolu	3.1, 3.1.1	3.1, 3.1.1
Viacnásobné pripojenie a manažment pripojení	×	✓
Vysoká dostupnosť	✓	✓
Konfiguračný súbor	✓	✓
Diagnostické logovanie	✓	✓
Manuálne odosielanie správ	×	✓
Odosielanie správ pomocou skriptov	✓	✓
Automatizované testovanie	✓	✓
Preddefinované prihlásenie na topic	✓	✓
Prihlasovanie na topicy pomocou skriptov	✓	✓
Manažment prihlásení na topicy	×	✓
Náhľad prijatých správ	✓	✓
Vyhľadávanie a prehľadávanie správ	×	✓
Formátovanie a dekódovanie obsahu správ	✓	✓
Prehľad topicov, filtrovanie a vyhľadávanie	×	✓
Vytváranie audit logu správ	✓	✓
Prehľadávanie audit logu správ	×	✓
Grafy	×	✓
Štatistika	×	✓
Kontrola aktualizácií pri štarte	×	✓

Tabuľka 3: Porovnanie funkcionality nástrojov mqtt-spy a mqtt-spy-daemon[2].

softvéru, preto v práci popisujeme práve ten, no existuje veľa ďalších podobných nástrojov na zaznamenávanie a monitorovanie aktivity.

9 Návrh a implementácia prototypu

V predchádzajúcich sekciách sme popísali protokol MQTT a pozreli sme sa na možnosti a funkcionalitu MQTT brokerov a klientov. Pri konkrétnom návrhu prototypu sa budeme ďalej zaoberať niekoľkými problémami:

- **point-to-point komunikácia**

[2] Zdroj: `https://github.com/eclipse/paho.mqtt-spy/wiki`

  - návrh komunikácie (požiadavka-odpoveď) dvoch mikroslužieb,
- **návrh mikroslužby na vizualizáciu a ovládanie**
  - výber vhodných technológií,
  - návrh API pre komunikáciu s inými mikroslužbami,
  - implementácia,
- **návrh mikroslužby uchovávajúcej dáta**
  - výber databázového systému,
  - návrh API pre komunikáciu s inými mikroslužbami,
  - implementácia,
- **bezpečnosť**
  - návrh konfigurácie MQTT brokera s cieľom zvýšiť bezpečnosť,
  - návrh hierarchie topicov a obmedzenie prístupových práv klientom.

9.1 Komunikácia dvoch mikroslužieb

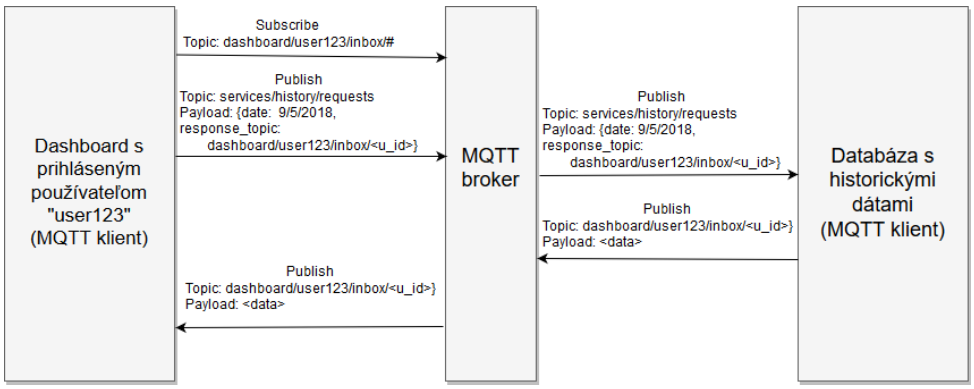
Senzory pri odosielaní správ o nameraných hodnotách nepotrebujú nutne dostať odpoveď o prijatí správy inými klientmi, no je veľa iných situácií, kedy je žiaduce poznať nielen kód odpovede na požiadavku, ale aj spätne odoslať dáta. Jednotlivé komponenty IoT systému potrebujú medzi sebou komunikovať a dostávať odpovede. MQTT nemá koncept požiadavky a odpovede ako REST, no existujú spôsoby, ako tento model simulovať. Pri tom treba brať do úvahy to, že odpoveď má byť doručená iba odosielateľovi požiadavky. Autor práce [4] popisuje dve metódy simulácie HTTP odpovede na požiadavku:

- **Odpoveď v tele správy:** V prvom prístupe prebehne najprv publish-subscribe komunikácia na doručenie identifikátora odpovede odosielateľovi požiadavky. Odosielateľ sa pripojí na odber správ z topicu s týmto identifikátorom, až potom odošle správu na topic. Klient, ktorý je na danom topicu prihlásený obdrží z brokera správu a kód odpovede odošle do spomínaného topicu s identifikátorom odpovede, na ktorý je pôvodný odosielateľ správy prihlásený.
- **Odpoveď v názve topicu:** Druhý prístup spočíva v zahrnutí kódu odpovede do hierarchie topicu. Odosielateľ sa najprv pripojí na odber správ z topicu s kódom odpovede, potom urobí publish správy. Napríklad, ak odosielateľ chce poslať správu do topicu *upjs/jesenna/P09/teplota*, najprv sa prihlási na odber z topicu *upjs/jesenna/P09/teplota/200*. Klient, ktorý je prihlásený na odber z topicu *upjs/jesenna/P09/teplota* dostane od brokera správu, ktorú odoslal odosielateľ a pošle svoju odpoveď do topicu *upjs/jesenna/P09/teplota/200*. Takto môže odosielateľ správy dostať od príjemcu kód odpovede na svoju požiadavku. Najväčšou nevýhodou tohto prístupu je, že odosielateľ sa pred poslaním správy potrebuje pripojiť na všetky možné kódy odpovedí. Tento prístup spolu s rapídnym nárastom počtu IoT zariadení nie je v súlade s ich obmedzenými zdrojmi, keďže kvôli jednej správe musí prebehnúť množstvo prihlásení a odhlásení z topicov.

Pri implementácii komunikácie dvoch služieb využijeme princíp podobný prvému z vyššie spomínaných. Keďže je žiadúce minimalizovať počet poslaných správ, topic, do ktorého sa zašle odpoveď, môže byť zahrnutý v tele správy, ktorú posiela odosielateľ. Topic by mal spĺňať niekoľko obmedzení:

- odpoveď si z neho môže prečítať iba mikroslužba posielajúca požiadavku, ak ide o dáta špecifické pre konkrétneho používateľa, tak k nim môže mať prístup len on,
- právo na čítanie z topicu, do ktorého sa posiela požiadavka so zahrnutým topicom pre odpoveď, by mala mať iba cieľová mikroslužba,
- hierarchická štruktúra topicu by mala obsahovať jedinečný identifikátor vygenerovaný pre konkrétnu požiadavku.

Implementáciu komunikácie a pravidiel pre prístupy k topicom budeme testovať na prototype. Obrázok 4 zobrazuje príklad priebehu komunikácie dvoch služieb.



Obr. 4: Príklad simulácie point-to-point komunikácie

9.2 Hierarchia topicov a prístupové práva

V tejto sekcii si popíšeme jednu z možností, ako efektívne rozdeliť hierarchiu topicov, aby boli zabezpečené základné požiadavky pre prístupy do jednotlivých topicov pri komunikácii mikroslužieb. Princíp spočíva v pridelení prefixu topicov každej mikroslužbe a rozdelení nasledujúcej úrovne na 3 schránky.

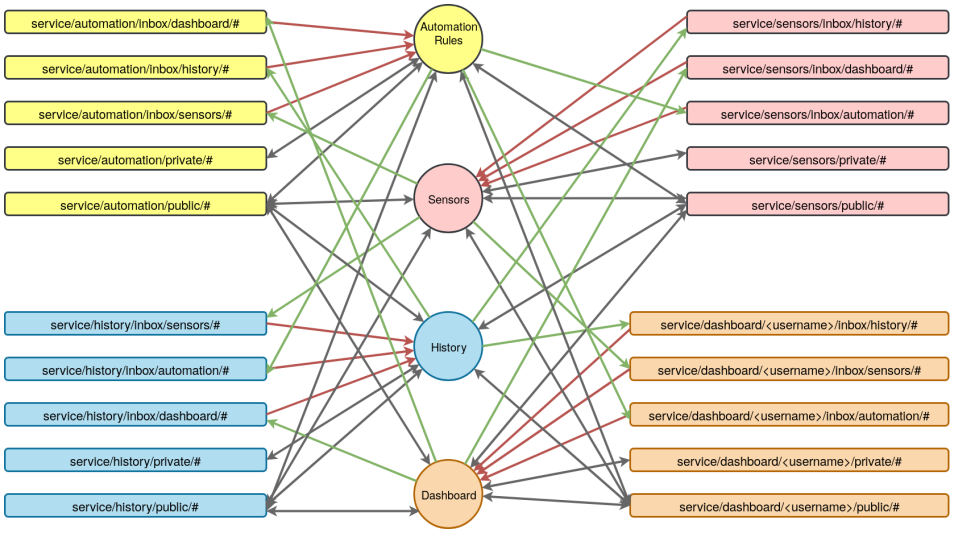
Prefix prislúchajúci mikroslužbe je service/<*service_username*>, kde pre každú mikroslužbu <*service_username*> nahradíme používateľským menom, ktorým sa pripája MQTT klient danej mikroslužby na MQTT broker.

V ďalšej úrovni rozdelíme topicy do troch skupín, a to:

- **service/<*service_username*>/public** - topicy súvisiace s mikroslužbou <*service_username*>, ktoré nevyžadujú súkromný prístup, práva na čítanie a zápis do nich majú všetky mikroslužby,

– **service/<*service_username*>/private** - všetky topicy začínajúce týmto prefixom sú prístupné na zápis aj čítanie iba mikroslužbe s menom <*service_username*>,
– **service/<*service_username*>/inbox** - topicy slúžiace na *point-to-point* komunikáciu dvoch mikroslužieb. Právo čítania z nich má len mikroslužba s menom <*service_username*>. Naopak, práva na zápis do topicov s prefixom service/<*service_username*>/inbox/<*service2_username*> má výhradne mikroslužba s menom <*service2_username*>

Na obrázku 5 je znázornený príklad prístupov mikroslužieb (kvôli jednoduchosti v obmedzenom počte) do jednotlivých topicov. V obdĺžnikoch sú znázornené topicy, v kruhoch mikroslužby. Šípky sivej farby predstavujú prístup mikroslužby do topicov daným prefixom, ktorý umožňuje čítanie aj zápis. Červená šípka znázorňuje právo čítania a zelená právo zápisu.



Obr. 5: Graf prístupov k topicom.

Nasledujúci postup popisuje príklad komunikácie medzi mikroslužbami A a B. Nech služba A požaduje dáta od služby B. Mikroslužba B je prihlásená na odber správ z topicu service/B/inbox/A/requests, kde očakáva požiadavky.

1. Mikroslužba A vygeneruje identifikátor <uuid> na jednorázové použitie pre konkrétnu odpoveď od služby B.
2. Mikroslužba A sa prihlási na odber správ z daného topicu.
3. Mikroslužba A posiela do topicu service/B/inbox/A/requests požiadavku v tvare

```
{
  "request": {...},
  "response_topic": "service/A/inbox/B/<uuid>"
}.
```

4. Mikroslužba B spracuje požiadavku obsiahnutú v *request* a odpoveď odošle do topicu service/A/inbox/B/<uuid>.
5. Mikroslužbe A je odpoveď z topicu service/A/inbox/B/<uuid> doručená.
6. Mikroslužba A sa odhlási z odberu správ z topicu service/A/inbox/B/<uuid>.

Je zjavné, že kroky 2 a 6 sa nemusia vykonávať, ak bude nepretržite mikroslužba A prihlásená na odber z topicov service/A/inbox/B/#. Každý z prístupov má svoje pozitíva a negatíva. Ak je mikroslužba A prihlásená na odber z topicov service/A/inbox/B/#, vyhneme sa prihlasovaniu a odhlasovaniu z topicu pri každej komunikácii. Na druhej strane ak sa mikroslužba pripojí a odpojí z odberu iba z konkrétneho topicu, môžme sa vyhnúť prípadným problémom s nevyžiadanými správami v rôznych topicoch s prefixom service/A/inbox/B.

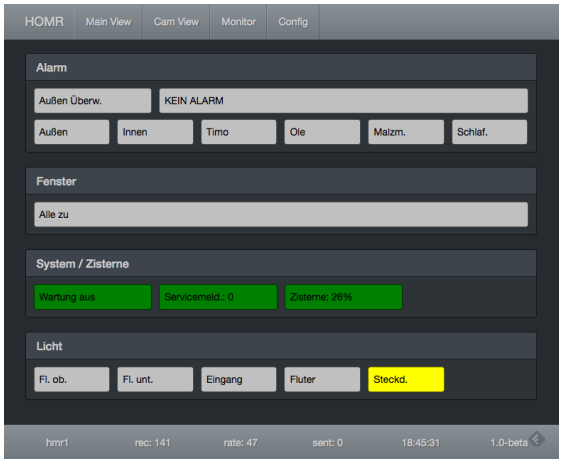
9.3 Dashboard

Teraz sa popíšeme zopár detailov návrhu jednotlivých komponentov sytému. Medzi najdôležitejšie časti integračného systému pre Smart Home bez pochýb patrí používateľské rozhranie. Je nevyhnutné, aby mohol používateľ kontrolovať a ovládať všetky prvky inteligentného domova, a to s čo najvyšším komfortom. V tejto časti sa pozrieme bližšie na návrh a implementáciu tohto komponentu.

Najskôr ale podotkneme, že existujú podobné komponenty na vizualizáciu dát získaných z IoT siete. Toto sú niektoré z nich:

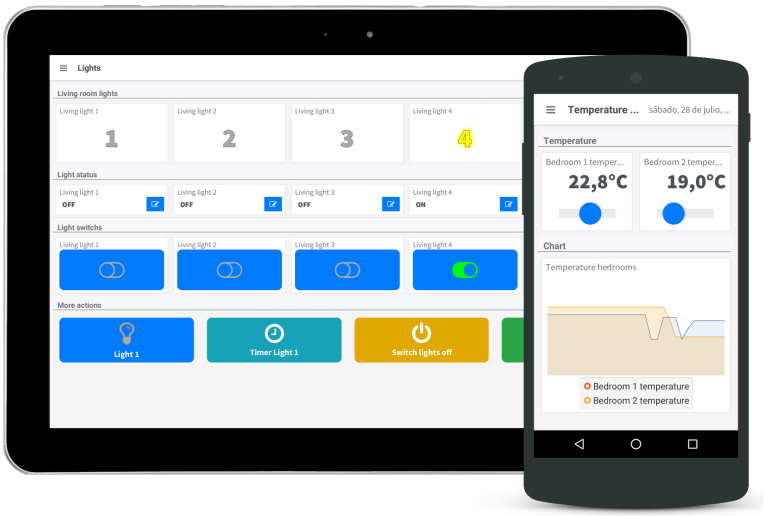
– **HOMR-REACT** je html5 aplikácia, ktorá poskytuje jednoduché zobrazovanie a prepínače pre Smart Home. Po otvorení aplikácie vo webovom prehliadači sa vyžaduje zadanie URL s konfiguračným súborom. Súbor je vo formáte JSON a nachádzajú sa v ňom informácie potrebné na pripojenie na MQTT broker a definícia widgetov nachádzajúcich sa na jednotlivých obrazovkách aplikácie. Toto riešenie ale neposkytuje žiadnu formu zabezpečenia, autentifikáciu používateľa ani zobrazovanie grafov historických dát. Obrázok 6 zobrazuje jednu z možných obrazoviek tejto aplikácie.

[3] Zdroj: `https://github.com/klauserber/homr-react`



Obr. 6: HOMR-REACT[3].

– **HelloIoT** je Java aplikácia, ktorá môže byť spustená na operačnom systéme Windows, MacOS, Linux alebo Android. Môže byť použitá ako MQTT klient na odosielanie a odber správ z a do topicov.



Obr. 7: HelloIoT[4].

[4] Zdroj: `https://github.com/adrianromero/helloiot`

Podobných aplikácií existuje samozrejme omnoho viac. Je viacero dôvodov, prečo žiadna z nich nie je pre nás vyhovujúce. Buď sú to opäť komplexné aplikácie s vlastnou databázou určené napríklad pre mobilné zariadenia alebo sú to na druhej strane veľmi jednoduché riešenia s úplne základnou funkcionalitou zobrazovania, ktorá nezahŕňa možnosti autentifikácie či prehliadania ľubovoľných intervalov historických dát. Túto funkcionalitu považujeme za nevyhnutnú.

Hlavným cieľom je implementovať dashboard ako skupinu statických stránok, ktoré nevyužívajú žiadne lokálne úložisko pre konfiguráciu zobrazeného obsahu, čo znamená, že používateľ sa môže prihlásiť z ľubovoľného zariadenia a zobrazený obsah je závislý len od prihlasovacích údajov používateľa. Rovnako je možné, aby sa z jedného zariadenia prihlasovali viacerí používatelia a každému sa zobrazí jeho vlastný obsah. Lokálna konfigurácia zahŕňa iba informácie potrebné na pripojenie na MQTT broker, akými sú adresa a port, na ktorom MQTT broker počúva. Ďalším z predpokladov je, že nami navrhnutý dashboard musí byť schopný fungovať v ľubovoľnej MQTT sieti bez nutnosti ďalších komponentov (samozrejme s obmedzenou funkcionalitou, napríklad bez možnosti zobrazenia historických dát, ak nemáme k dispozícii mikroslužbu, ktorá ich uchováva), no je schopný komunikovať s ďalšími mikroslužbami len prostredníctvom MQTT brokera.

9.3.1 Implementácia

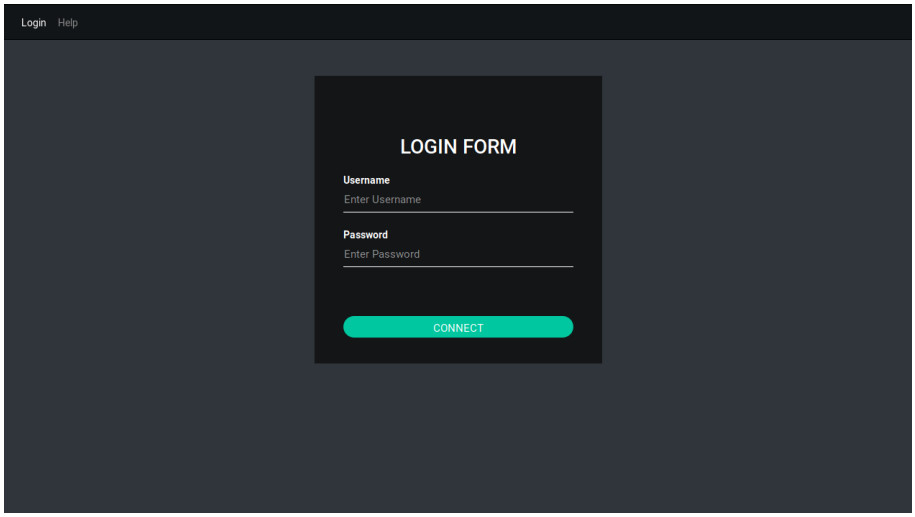
V tejto časti popíšeme výber technológií pre mikroslužbu dashboard a priblížime si niektoré detaily jej implementácie, rovnako ako aj spôsob jej použitia. Pre implementáciu mikroslužby dashboard sme zvolili programovací jazyk JavaScript a jeho knižnice React a Redux.

React.js je v posledných rokoch jednou z najčastejších volieb pri výbere technológií pre používateľské prostredie webovej aplikácie. Dôraz sa pri vývoji kladie na tvorbu komponentov, každý z nich má svoju množinu premenných *state* a *props*. Tie uchovávajú zvyčajne dáta potrebné na vykreslenie daného komponentu. Rozdielom medzi nimi je, že parametre *props* sú nastavené rodičovským komponentom a samotný komponent ich už nemôže meniť. Na druhej strane parametre *state* sú komponentom modifikované. Pri zmene ktorejkoľvek z týchto premenných sa komponent aktualizuje a znova vykreslí. Knižnica React.js je vhodnou voľbou pri aplikáciách, ktoré uchovávajú a modifikujú väčšie množstvo dát.

React Redux umožňuje vytvoriť centrálny *store*, ktorý uchováva a sprístupňuje *state* pre všetky komponenty. Vďaka nemu je práca s dátami v rámci celej aplikácie omnoho jednoduchšia.

Teraz sa pozrieme na návrh a implementáciu mikroslužby dashboard. Jej použitie je vcelku jednoduché. Po otvorení stránky sa ako prvá zobrazí stránka prihlásenia (obrázok 8) na MQTT broker. Dashboard môže byť použitý aj v kombinácii

s brokerom, ktorý nevyžaduje autentifikáciu prihláseného používateľa. No aj v takom prípade je potrebné zadať používateľské meno, pretože s ním súvisí samotný obsah a konfigurácia dashboardu.



Obr. 8: Stránka prihlásenia.

Po prihlásení používateľa nasleduje automatické prihlásenie na odber správ zo špeciálnych topicov:

- service/dashboard/private/<*username*>/config/widgets
- service/dashboard/private/<*username*>/config/dashboards
- service/dashboard/inbox/<*username*>/#

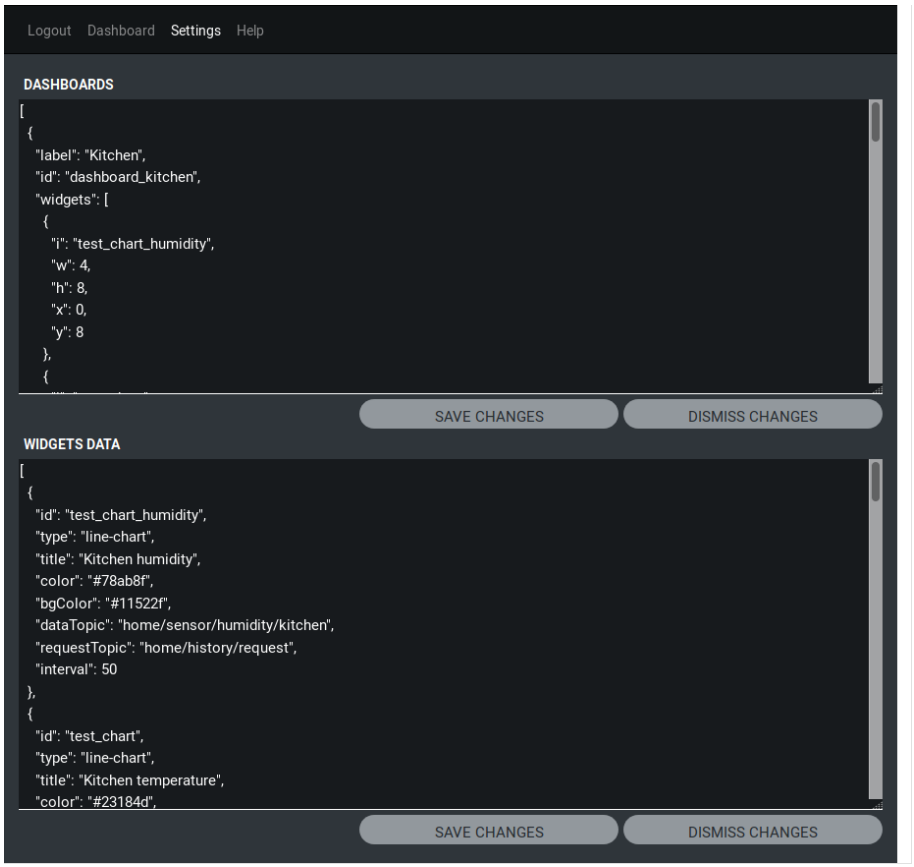
Prvé dve z nich sú určené na uchovanie konfigurácie pre widgety a jednotlivé obrazovky, medzi ktorými môže používateľ prepínať. Do týchto topicov má prístup na čítanie aj zápis iba mikroslužba dashboard a iba v prípade, že je prihlásený práve používateľ s daným používateľským menom. Skupina topicov service/dashboard/inbox/<*username*>/# slúži na komunikáciu s ostatnými mikroslužbami, viac o tom sme popísali v sekcii 9.2.

Po prvom prihlásení je dashboard prázdny. V hornej časti obrazovky vidí prihlásený používateľ štyri možnosti:

- Klinutím na ***Logout*** sa MQTT klient odpojí od brokera a všetky dáta používateľa uchovávané v *store* sa odstránia. Následne sa dostane späť na obrazovku prihlásenia.
- Možnosť ***Dashboard*** zobrazuje hlavný obsah (príklad na obrázku 11), teda samotné dashboardy s widgetmi.

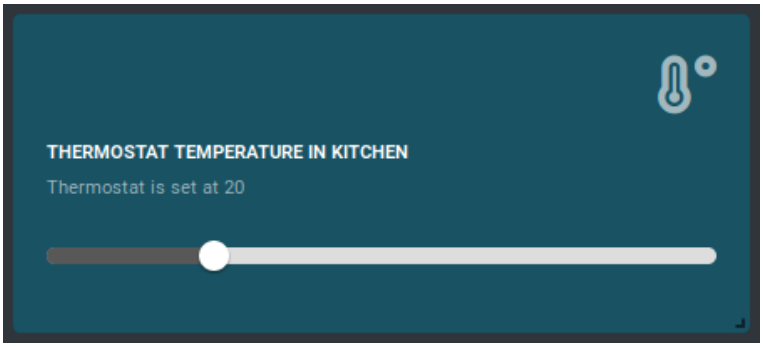
– Obsah dashboardov môže používateľ nakonfigurovať po kliknutí na možnosť ***Settings***.

– Na stránke ***Help*** sa nachádza návod a ukážky konfigurácie dashboardov.



Obr. 9: Stránka konfigurácie.

Obrázok 9 zobrazuje konfiguráciu widgetov a dashboardov vo formáte JSON. V spodnej časti obrázka sa nachádza konfigurácia widgetov. Konfigurácia pozostáva zo zoznamu JSON-ov pre jednotlivé widgety. Každý z týchto widgetov môže byť použitý vo viacerých dashboardoch. Obrázok 10 zobrazuje jednoduchý widget.



Obr. 10: Príklad widgetu.

Na konfiguráciu widgetu je potrebný nasledujúci JSON:

```
{
  "id": "thermostat_kitchen",
  "type": "slider",
  "icon": "temperature",
  "title": "Thermostat Temperature in Kitchen",
  "info": "Thermostat is set at ",
  "color": "#185263",
  "data": "service/sensors/kitchen/thermostat",
  "min": 10,
  "max": 50
}
```

Obsahuje nasledujúce parametre:

- **id** - unikátny identifikátor widgetu (v rámci daného zoznamu),
- **type** - typ widgetu (v tomto prípade widget na výber hodnoty z rozsahu min - max),
- **icon** - názov ikony, ktorú chceme zobraziť na widgete (z ponuky bezplatných ikon z `https://fontawesome.com/`),
- **title** - názov zobrazený na widgete,
- **info** - doplňujúci popis,
- **color** - farba pozadia,
- **data** - topic, zdroj dát z/do ktorého sa prijímajú/odosielajú dáta pre widget,
- **min** - minimálna hodnota rozsahu,
- **max** - maximálna hodnota rozsahu.

Druhým krokom konfigurácie je vytvorenie zoznamu dashboardov a ich obsah. Konfigurácia jedného dashboardu, ktorý obsahuje tri widgety, môže vyzerať následovne:

```
{
  "label": "Kitchen",
  "id": "dashboard_kitchen",
  "widgets": [
        {
          "i": "kitchen_temperature",
          "w": 3,
          "h": 5,
          "x": 0,
          "y": 0
        },
        {
          "i": "kitchen_humidity",
          "w": 3,
          "h": 5,
          "x": 5,
          "y": 0
        },
        {
          "i": "kitchen_light",
          "w": 2,
          "h": 5,
          "x": 3,
          "y": 0
        }
  ]
}
```

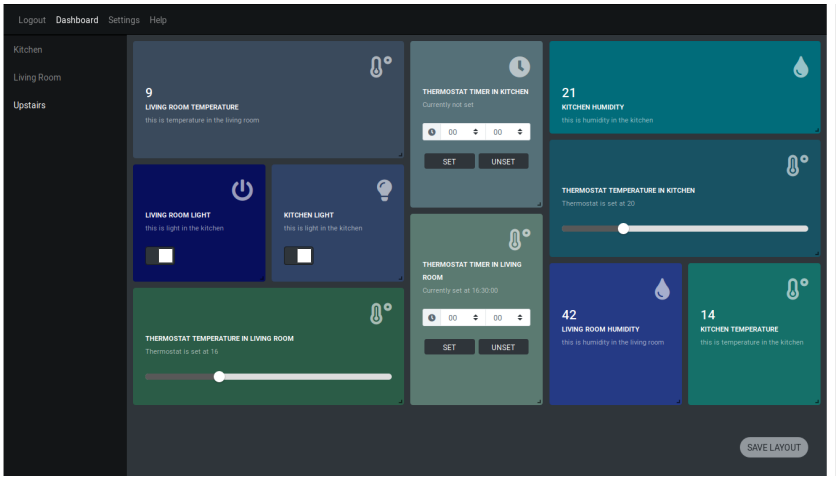
Konfigurácia obsahuje:

- **label** - názov, ktorý sa zobrazuje v ponuke dashboardov,
- **id** - unikátny identifikátor dashboardu (v rámci zoznamu),
- **widgets** - zoznam widgetov, ktoré sa zobrazujú na dashboarde, každý widget obsahuje nasledujúce parametre:
  - **i** - unikátny identifikátor widgetu (zo zoznamu widgetov)
  - **w** - šírka widgetu,
  - **h** - výška widgetu,
  - **x** - súradnica x ľavého horného rohu,
  - **y** - súradnica y ľavého horného rohu,

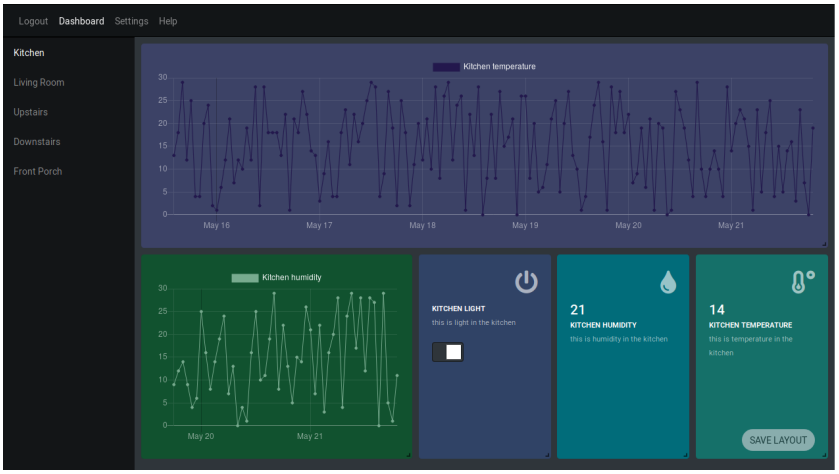
pričom hodnoty parametrov *w, h, x* a *y* predstavujú počet dielikov mriežky dashboardu. Nie je potrebné ich manuálne počítať a nastavovať. Pridaný widget môže mať predvolené hodnoty:

- **w** = 4,
- **h** = 4,
- **x** = 0,
- **y** = 99 (alebo ľubovoľné „veľké“ číslo),

vďaka čomu sa widget zobrazí pri ľavom okraji na najvyššom možnom mieste, keďže widgety v mriežke sú automaticky presúvané na najvyššiu možnú voľnú pozíciu so zachovaním y-súradnice. Následne vie používateľ zmeniť pozíciu widgetu systémom *drag and drop* a tiež zmeniť jeho veľkosť a tvar ťahaním značky v pravom dolnom rohu. Toto riešenie umožňuje umiestniť widgety podľa želania.



Obr. 11: Príklad dashboardu.



Obr. 12: Príklad dashboardu 2.

Na obrázku 11 je zobrazený dashboard zložený z jednoduchých widgetov zobrazujúcich dáta a takých, ktoré umožňujú nastaviť hodnotu času, vypínača alebo vybrať hodnotu z nejakého rozsahu. Obrázok 12 zobrazuje dashboard, ktorý obsahuje aj grafy historických dát. Tieto dáta boli pred vykreslením widgetu aplikáciou vyžiadané podľa konfigurácie, napríklad:

```
{
  "id": "test_chart_humidity",
  "type": "line-chart",
  "title": "Kitchen humidity",
  "color": "#78ab8f",
  "bgColor": "#11522f",
  "dataTopic": "service/sensors/kitchen/humidity",
  "requestTopic": "service/history/inbox/request",
  "interval": 50
},
```

kde:

- **color** - farba čiar grafu,
- **bGolor** - farba pozadia widgetu,
- **dataTopic** - topic, z ktorého dáta sú ukladané do databázy,
- **requestTopic** - topic, v ktorom mikroslužba uchovávajúca historické dáta očakáva požiadavky,
- **interval** - počet hodnôt, ktoré chceme zobraziť.

9.4 Mikroslužba uchovávajúca historické dáta

Na to, aby sme dokázali zobrazovať historické dáta, potrebujeme mikroslužbu, ktorá ich bude uchovávať. Zobrazenie historických dát je jednou z požadovaných funkcií integračného systému pre Smart Home. Keďže inteligentné domácnosti často pozostávajú z množstva senzorov, meračov, vypínačov, regulátorov a ďalších podobných zariadení, používateľ by mal mať možnosť prezerať si ich namerané, či nastavené hodnoty v požadovaných časových intervaloch. Preto sa v tejto sekcii pozrieme na niekoľko databázových systémov.

Je zjavné, že použitie relačnej databázy pri takýchto špecifických dátach nie je vhodnou voľbou. Operácie vykonávané nad databázou zahŕňajú prevažne iba chronologické ukladanie dát do tabuliek a dopyty, ktoré vrátia plynulý úsek dát z tabuľky podľa kľúča, ktorým je časová pečiatka. Preto nemusíme dbať na ACID vlastnosti relačných databáz.

Mimoriadny rozmach zažívajú v posledných rokoch databázy špecializované na ukladanie časových radov dát, ktoré sú optimalizované práve na zápis veľkého množstva dát a dopytovanie na riadky tabuľky podľa časového intervalu. Databázy zamerané na časové rady sú čoraz využívanejšie aj z dôvodu rozmachu oblasti internetu vecí, kde množstvo zariadení zaznamenáva rôzne veličiny. Sú však využívané aj v mnohých iných oblastiach na zaznamenávanie rôznych udalostí a údajov. V nasledujúcich sekciách niektoré z týchto databáz popisujeme.

9.4.1 InfluxDB je *open-source* databázový systém, ktorý má podporu pre všetky operačné systémy. Podporuje veľké množstvo programovacích jazykov. Je optimalizovaný pre zápis veľkého množstva dát a funguje veľmi dobre v konkurentnom prostredí. Patrí medzi NoSQL databázy a umožňuje rýchle zmeny databázovej schémy. práca s InfluxDB je veľmi jednoduchá, nie je potrebná konfigurácia na to, aby sme mohli okamžite ukladať dáta.

9.4.2 TimescaleDB je takisto *open-source*, tiež umožňuje priamu integráciu do aplikácie v mnohých programovacích jazykoch. Veľkou výhodou tejto databázy je, že podporuje jazyk SQL, preto nie je potrebné učiť sa nový jazyk. Vznikla ako rozšírenie databázového systému PostgreSQL so zameraním na dáta časových radov.

9.4.3 OpenTSDB je starším databázovým systémom, ako predchádzajúce dve. Používateľom sľubuje schopnosť uloženia stoviek miliárd riadkov dát v distribuovaných inštanciách TSD serverov. Ponúka vysoký výkon až do desiatok miliónov zápisov za sekundu.

9.4.4 Graphite je zo spomínaných systémov najstarší. Ponúka nástroj na monitorovanie, ktorý ukladá numerické časové rady dát a zobrazuje ich na vyžiadanie vo webovom rozhraní. Používajú ho veľké spoločnosti, akými sú Booking.com, Reddit a Github.

9.4.5 extremeDB je komerčný databázový systém, ktorý vďaka svojej flexibilite umožňuje stanoviť prioritu spomedzi výkonu, šetrenia nákladov, energie alebo pamäte. Je to hybridný systém, ktorý kombinuje pozitíva databáz uložených na disku aj tých, ktoré svoje dáta uchovávajú primárne v hlavnej pamäti. Má podporu pre všetky platformy, je vhodný pre *cloud* aj *egde computing* a poskytuje okrem iných výhod aj podporu pre časové rady dát.

9.4.6 RedisTimeSeries je modul databázového systému Redis, ktorý pridáva podporu pre časové rady. Umožňuje nájsť minimum, maximum, priemer, súčet, rozsah a počet vybranej hodnoty v časovom intervale hodnôt vďaka podpore agregačných funkcií.

Okrem spomínaných možností je k dispozícii množstvo ďalších open-source i komerčných databázových systémov, medzi ktoré patria napríklad Apache Cassandra, Atlas, Beringei, Elasticsearch, IoTDB a iné. Výber databázy závisí samozrejme aj od zariadenia, na ktorom bude mikroslužba uchovávajúca historické dáta bežať.

Pri implementácii prototypu budeme pracovať s viacerými možnosťami. Pri testovaní komunikácie mikroslužby poskytujúcej historické dáta a mikroslužby, ktorá poskytuje vizualizáciu a ovládanie Smart Home systému sme testovali verziu bez databázy, čo znamená, že všetky dáta boli uchovávané iba v operačnej

pamäti počítača počas behu aplikácie. To samozrejme odporuje princípu ukladania historických dát, ale pre testovanie návrhu architektúry a komunikácie mikroslužieb bola táto jednoduchá verzia vhodná. V ďalšej verzii použijeme jednu z vyššie uvedených databázových systémov pre časové rady dát - a to TimescaleDB, z dôvodu jeho jednoduchého použitia.

10 Záver

V práci sme popísali a zdôvodnili návrh integračného riešenia využívajúceho architektúru podobnú mikroslužbám. Táto architektúra má mnoho výhod pri súčasnom stave vývoja oblasti internetu vecí. Ďalej sme odôvodnili výber komunikačného protokolu MQTT a bližšie sme sa pozreli na jeho možnosti. Spomenuli sme niekoľko z mnohých schopností samotného protokolu ale aj vybraného MQTT brokera, ktoré pri implementácii infraštruktúry riešenia môžeme využiť.

Popísali sme návrh hierarchie topicov a prístupových práv k nim. Zbežne sme predstavili zvolené technológie a hlavnú funkcionalitu mikroslužby dahsboard a spôsob jej použitia. Vyskúšali sme navrhnutú komunikáciu medzi komponentami dashboard a komponentom uchovávajúcim historické dáta. Pozreli sme sa na niekoľko databázových systémov uchovávajúcich časové rady dát.

Medzi nasledujúce ciele patrí samozrejme implementácia databázového komponentu, ktorý bude využívať jednu z predstavených databázových systémov pre časové rady dát a tiež rozšírenie možností výberu rôznych časových intervalov, keďže pri testovaní prvej verzie sme sa obmedzili na vyžiadanie iba posledných hodnôt vo zvolenom počte.

Takisto sa budeme zaoberať implementáciou ďalších typov widgetov, aby poskytli používateľovi čo najväčšie možnosti využitia. Zároveň by bolo z hľadiska používateľského komfortu vhodné implementovať príjemnejší spôsob pridávania widgetov do dahsboardu, ktorý by nevyžadoval ručné písanie konfigurácie vo formáte JSON.

Preskúmame možnosť využitia viacerých brokerov. Popíšeme ďalšie komponenty systému, ktoré by mali pokrývať funkcionalitu napríklad komunikácie so zariadeniami rozličnými protokolmi či automatizačných pravidiel.

Literatúra

1. BUYYA, Rajkumar; DASTJERDI, Amir Vahid (ed.). Internet of Things: Principles and paradigms. Elsevier, 2016.
2. SUN, Long; LI, Yan; MEMON, Raheel Ahmed. An open IoT framework based on microservices architecture. China Communications, 2017, 14.2: 154-162.
3. GUINARD, Dominique; TRIFA, Vlad. Building the web of things: with examples in node. js and raspberry pi. Manning Publications Co., 2016.
4. MANANDHAR, Srijan. MQTT based Communication in IoT. 2017.
5. LIGHT, Roger A. Mosquitto: server and client implementation of the MQTT protocol. The Journal of Open Source Software, 2017, 2.13: 265.
6. UVIASE, Onoriode; KOTONYA, Gerald. Iot architectural framework: connection and integration framework for iot systems. arXiv preprint arXiv:1803.04780, 2018.

7. LEWIS, James; FOWLER, Martin. Microservices. martinfowler. com, 2014.
8. THÖNES, Johannes. Microservices. IEEE software, 2015, 32.1: 116-116.
9. FETTE, Ian; MELNIKOV, Alexey. The websocket protocol. 2011.
10. BOX, Don, et al. Simple object access protocol (SOAP) 1.1. 2000.
11. FIELDING, Roy Thomas. REST: architectural styles and the design of network-based software architectures. Doctoral dissertation, University of California, 2000.
12. BUNA, Samer. Learning GraphQL and relay. Packt Publishing Ltd, 2016.
13. FIELDING, Roy, et al. Hypertext transfer protocol–HTTP/1.1. 1999.
14. PAHO, Eclipse. Eclipse Paho-MQTT and MQTT-SN software. [online] (citované 03.01.2020): `http://www.eclipse.org/paho`.