Integračný softvér pre smart home inšpirovaný mikroslužbami
Analýza a návrh riešenia

Patrícia Szepesiová

1Im, 2018-2019

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 jednoduchú implementáciu, 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

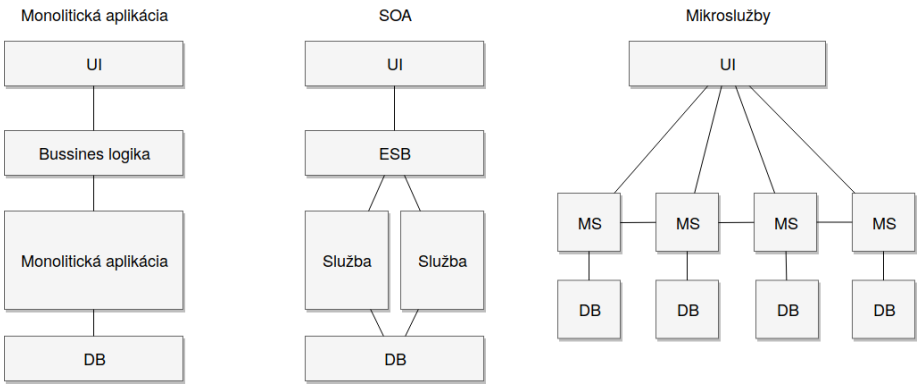
V tejto sekcii popíšeme a odôvodníme výber architektúry mikroslužieb. Jeden zo spôsobov rozdelenia architektúr je na monolitické a distribuované. Medzi distribuované architektúry patria už spomínané mikroslužby a SOA. Voľba distribuovanej architektúry je jednoznačná v prípade integračného softvéru pre 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. V nasledujúcej sekcii popíšeme návrh komunikácie medzi mikroslužbami.

5 Protokol MQTT

V tejto sekcii popíšeme protokol MQTT, jeho výhody, nevýhody a prečo sme sa rozhodli pre jeho použitie. MQTT je aplikačný protokol, ktorý využíva TCP protokol transportnej vrstvy. Je jednoduchý a nenáročný, preto je využívaný v oblasti internetu vecí, kde je často problém s obmedzenými zdrojmi. Jeho cieľom je spoľahlivé doručenie v nespoľahlivej sieti alebo sieti s obmedzenými prostriedkami. Doručenie správy sa uskutočňuje použitím komunikačnej metódy publish-subscribe.

5.1 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.

5.2 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:

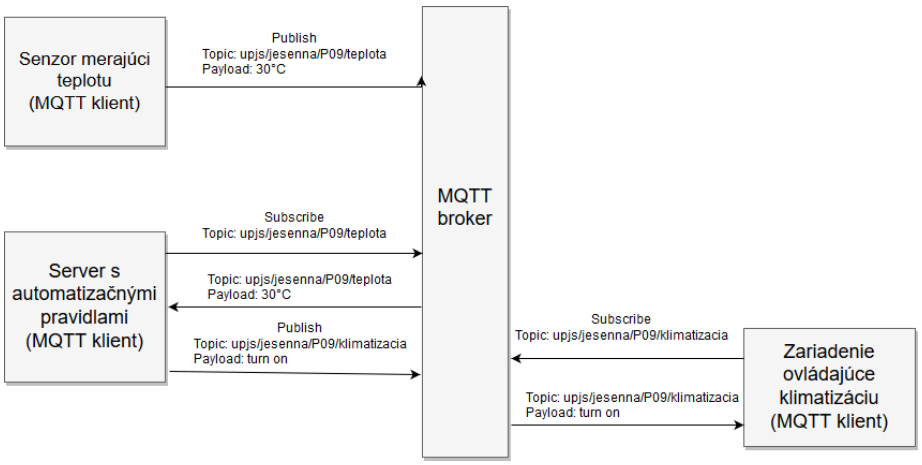
- **packetId** - unikátny identifikátor paketu,
- **topicName** - názov topicu,
- **qos** - *Quality of Service* definuje garanciu doručenia správy, môže mať tri hodnoty:
  - **0** - najviac raz,
  - **1** - aspoň raz,
  - **2** - práve raz,
- **retainFlag** - ak je hodnota nastavená na *true*, broker 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. Obrázok 2 znázorňuje príklad komunikácie pomocou protokolu MQTT.

5.3 Porovnanie MQTT a HTTP

V tejto časti popíšeme rozdiely medzi spôsobom komunikácie protokolom MQTT a REST-om [10], ktorý je často využívaný v klient-server architektúre pre distribuované prostredie a predstavuje jednoduché rozhranie na prenos špecifikovaných dát s využitím protokolu HTTP. Oba majú za cieľ výmenu dát medzi koncovými zariadeniami v sieti. Z pohľadu REST klient posiela požiadavky na server, ktorý je poskytovateľom služieb. V MQTT protokole sú klientmi publisher aj subscriber. V REST-e, služby ponúkané serverom sú identifikované pomocou URI identifikátora, ktorý predstavuje adresu sprostredkovanej služby. V MQTT prístupné služby môžu byť identifikované topicmi.

Podstatným rozdielom je, že REST klient musí poznať IP adresu servera, ktorého služby chce využiť. Na komunikáciu je potrebné, aby klient a server



Obr. 2. Príklad komunikácie protokolom MQTT

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. Pri použití publish-subscribe modelu stačí, aby sa vedeli všetky zariadenia pripojiť na MQTT broker.

Navyše 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.

V ďalšej časti sa pozrieme na niekoľko MQTT brokerov, ich vlastnosti a funkcie, ktoré ponúkajú.

6 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.

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

6.2 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*.

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.

6.3 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.

6.4 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.

6.5 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.

6.6 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.

7 Návrh riešenia

V predchádzajúcich sekciách sme popísali protokol MQTT a funkcionalitu MQTT brokerov. Pri konkrétnom návrhu riešenia sa budeme ďalej zaoberať niekoľkými problémami. Keďže pri komunikácii protokolom MQTT klienti komunikujú vždy prostredníctvom brokera, musíme riešiť návrh „point-to-point“ komunikácie dvoch klientov. Pritom je potrebné brať ohľad na zabezpečenie. Bezpečnosť komunikácie medzi klientom a brokerom je zabezpečovaná samotným protokolom. Chceme však zabezpečiť aj to, aby medzi sebou mohli komunikovať len určité mikroslužby, keďže z bezpečnostného hľadiska nemusí byť povolenie všetkých prepojení medzi nimi vhodné. Navyše pri informáciach špecifických pre používateľov treba zamedziť prístupu používateľa k dátam iných používateľov.

7.1 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:

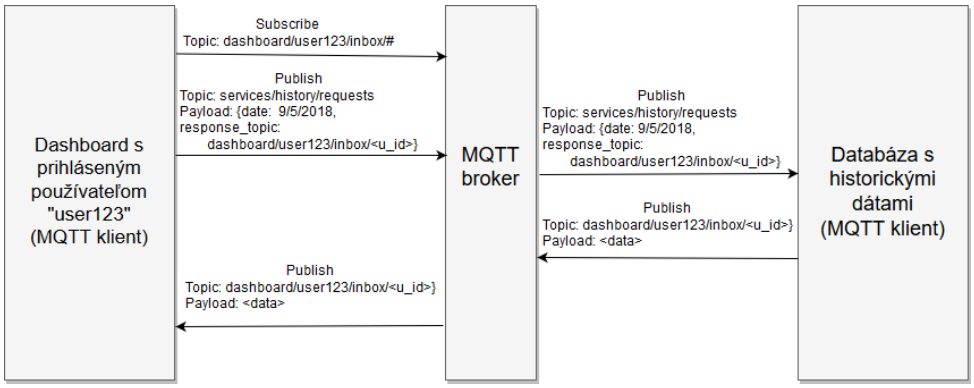
- **Odpoveď v tele správy:** V prvom prístupe prebehne najprv publish-subscribe komunikácia na doručenie identifikátora odpovede odosielateľovi požiadavky. Odosielateľ sa pripojí na odber správ z topicu s týmto identifikátorom, až potom odošle správu na topic. Klient, ktorý je na danom topicu prihlásený 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 3 zobrazuje príklad priebehu komunikácie dvoch služieb.



Obr. 3. Príklad simulácie point-to-point komunikácie

8 Záver

V článku sme popísali a zdôvodnili návrh integračného riešenia využívajúceho architektúru podobnú mikroslužbám. Táto architektúra ma mnoho výhod pri súčasnom stave vývoja oblasti internetu vecí. Ďalej sme popísali zvolený komunikačný protokol MQTT a jeho výhody. Spomenuli sme niekoľko z mnohých schopností samotného protokolu ale aj vybraného MQTT brokera, ktoré pri implementácii infraštruktúry riešenia môžeme využiť.

V ďalšej časti sa budeme zaoberať konkrétnym návrhom konfigurácie brokera, ako aj samotných mikroslužieb. Preskúmame aj možnosť využitia viacerých brokerov. Infraštruktúru riešenia a konkrétny návrh budeme testovať na prototype s niekoľkými mikroslužbami. Navrhneme a implementujeme *„dashboard“* - mikroslužbu na zobrazovanie a ovládanie. Konfigurovateľný zobrazovací

a ovládací panel by sme chceli implementovať ako skupinu statických stránok, ktorá sa po spustení pripojí na MQTT broker. Využijeme pri tom možnosť komunikácie s MQTT brokerom protokolom websocket. Súčasťou by mala byť samozrejme autorizácia používateľa aj zobrazovanie grafov historických dát, ktorých správu bude mať na starosti ďalšia mikroslužba. Pri implementácii dashboardu volíme JavaScript knižnice React a Redux. Ďalšie mikroslužby 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. FENG, Xinyang; SHEN, Jianjing; FAN, Ying. REST: An alternative to RPC for Web services architecture. In: 2009 First International Conference on Future Information Networks. IEEE, 2009. p. 7-10.5