Pokročilý sběr dat pro IoT Cloud s použitím standardizovaných komunikačních protokolů

Petr Velkoborský

Srpen 2021

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2020/2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Petr VELKOBORSKÝ**
Osobní číslo:	**A18B0554P**
Studijní program:	**B3918 Aplikované vědy a informatika**
Studijní obor:	**Kybernetika a řídicí technika**
Téma práce:	**Pokročilý sběr dat pro IoT Cloud s použitím standardizovaných komunikačních protokolů**
Zadávající katedra:	**Katedra kybernetiky**

Zásady pro vypracování

Seznamte se s nástrojem IoT Cloud [1].
Seznamte se se standardizovanými protokoly používanými v IoT (např. OPC UA [2], MQTT [3], JSON-RPC [4], REST).
Vyberte si alespoň jeden standardizovaný protokol, který budete implementovat pro sběr dat.
Navrhněte rozšíření konfigurace IoT Cloudu o parametry potřebné pro navázání komunikace s jednotlivými zdroji dat.
Naimplementujte samostatný, dostatečně univerzální, automaticky konfigurovatelný program, který bude aktivně získávat data pomocí standardizovaných protokolů ze vzdálených zařízení, zpracovávat je a ukládat je do IoT Cloudu.
Demonstrujte funkčnost programu na vhodně zvolených příkladech.

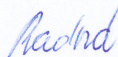
Rozsah bakalářské práce: **30-40 stránek A4**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná**

Seznam doporučené literatury:

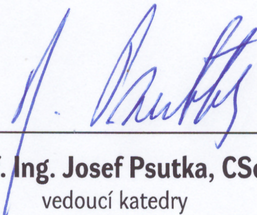
REX Controls: IoT Cloud. Referenční příručka. 2020.
OPC Unified Architecture Specification, online: https://opcfoundation.org/developer-tools/specifications-unified-architecture
MQTT Specification, online: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html
JSON-RPC Specification, online: https://www.jsonrpc.org/specification

Vedoucí bakalářské práce: **Ing. Tomáš Ausberger**
Výzkumný program 1

Datum zadání bakalářské práce: **15. října 2020**
Termín odevzdání bakalářské práce: **24. května 2021**



L.S.



Doc. Dr. Ing. Vlasta Radová
děkanka

Prof. Ing. Josef Psutka, CSc.
vedoucí katedry

V Plzni dne 15. října 2020

Prohlášení

Předkládám tímto k posouzení a obhajobě bakalářskou práci zpracovanou na závěr studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím odborné literatury a pramenů, jejichž úplný seznam je její součástí.

V Plzni dne 15.8.2021 ....

vlastnoruční podpis

Poděkování

Děkuji panu Ing. Tomáši Ausbergerovi za zajímavé téma práce, ochotu a pomoc při tvorbě workeru a také za vysvětlení, jak funguje IoT Cloud firmy REX Controls s.r.o.

Abstrakt

Tato bakalářská práce pojednává o vývoji a implementaci vlastního komunikačního rozhraní mezi IoT Cloudem a IoT zařízením s využitím komunikačního protokolu MQTT. Rozhraní je realizováno jako worker a bude zapisovat předaná data do databáze IoT Cloudu firmy RexControls s.r.o Celý program je automaticky konfigurovatelný, aby mohl pracovat nezávisle na uživateli a byl použitelný pro průmyslové využití.

Klíčová slova: IoT, MQTT, Cloud, Worker, publish/subscribe, broker, JSON.

Abstract

This bachelor's thesis deals with the development and implementation of a custom communication interface between IoT Cloud and IoT device using the MQTT communication protocol. The interface is implemented as a worker and will write the transmitted data to the IoT Cloud database of RexControls s.r.o. The whole program is automatically configurable to work independently of the user and to be usable for industrial applications.

Keywords: IoT, MQTT, Cloud, Worker, publish/subscribe, broker, JSON.

Obsah

1 Úvod **2**
- 1.1 Motivace . . . 3
- 1.2 Příprava řešení . . . 3

2 Teorie **5**
- 2.1 IoT . . . 5
- 2.2 Hrozby IoT . . . 5
- 2.3 Edge computing . . . 6
- 2.4 Cloud computing . . . 6
  - 2.4.1 Software as a Service . . . 6
  - 2.4.2 Infrastructure as a Service . . . 6
  - 2.4.3 Platform as a Service . . . 7
  - 2.4.4 Kritéria . . . 7
- 2.5 Cloudové úložiště . . . 7
  - 2.5.1 Storage as a Service . . . 7
- 2.6 PostgreSQL . . . 7
- 2.7 TimescaleDB . . . 8
- 2.8 HTTP REST . . . 8
- 2.9 JSON-RPC . . . 8
- 2.10 MQTT . . . 10
  - 2.10.1 Jak funje MQTT . . . 10
  - 2.10.2 MQTT broker . . . 10
  - 2.10.3 Typy MQTT zpráv . . . 10
- 2.11 Porovnání OPC UA a MQTT . . . 11

3 IoT Cloud **12**
- 3.1 Popis IoT cloudu firmy REX controls s.r.o . . . 12
- 3.2 Catalog API . . . 14

4 Realizace vlastního komunikačního rozhraní MQTT **15**
- 4.1 Experiment s MQTT . . . 15
- 4.2 Experiment s Cloudem . . . 16
- 4.3 Návrh architektury . . . 17
- 4.4 Části programu . . . 18
  - 4.4.1 Hlavní třída . . . 18
  - 4.4.2 Subscriber . . . 19

5 Výsledky **21**
- 5.1 Experiment . . . 21
- 5.2 Budoucí vývoj . . . 29

6 Závěr **30**

Kapitola 1

Úvod

Tato práce se věnuje vývoji komunikačního rozhraní mezi IoT zařízeními a IoT Cloudem. Toto rozhraní by mělo využívat standardizovaný protokol pro komunikaci s jednotlivými IoT zařízeními. Pro tyto účely jsme se rozhodli využít MQTT (Message Queuing Telemetry Transport), což je protokol typu publish/subscribe, který transportuje zprávy mezi zařízeními s pomocí protokolu ISO standard (ISO/IEC 20922). Dalšími vhodnými standardizovanými protokoly jsou například OPC UA, JSON-RPC, REST. Protokol MQTT byl zvolen zejména pro jeho jednoduchost a velké rozšíření. Toto rozhraní by mělo být realizováno jako worker. Worker je program, který dostane úkol a dál ho sám zpracovává, až docílí nějakého výsledku. Tento konkrétní worker je připravován pro komunikaci s IoT Cloudem firmy REX Controls s.r.o.

Navržené rozhraní by mělo obsluhovat MQTT protokol a zapisovat předaná data do databáze IoT Cloudu. Celý program by měl být co nejvíce automatizovaný, měl by se sám nakonfigurovat a pracovat nezávisle na uživateli. Zároveň by měl reagovat na změny provedené v databázi. Samozřejmě by měl být stabilní. Worker by měl být ideálně nasazený na stejném stroji jako daná databáze, ale není to nutností. Měl by sloužit jako jakási brána mezi klientem a databází. Díky tomuto workeru by mělo být ukládání dat na IoT Cloud bezpečnější a pro uživatele jednoduší.

Uživatel komunikuje pouze s workerem. Díky tomu nemusí zasahovat do databáze, zároveň nemusí vůbec řešit její nastavení a pouze odesílá zprávy ve správném tvaru na správnou adresu. Tyto zprávy jsou ve formátu JSON a obsahují informace důležité pro připojení a data, která mají být uložena. Uživatele se snažíme oddělit co nejvíce od složitějších věcí, neboť běžný uživatel nepotřebuje vědět co se děje na pozadí. Velký důraz je kladen na zabezpečení. Z toho důvodu jsou všechny zprávy šifrované, aby nedocházelo k žádnému úniku dat. Dalším prvkem zabezpečení jsou přístupové údaje, které každému uživateli umožňují přístup pouze do jeho vlastní databáze.

Konečný výsledek by měl být použitelný pro veřejnost. Měl by být interaktivní a jednoduchý pro uživatele. Příklady použití tohoto systému jsou například:

- Hlídání spotřeby energie jednotlivých budov a strojů v daném areálu.
- Monitorování vývoje teplot na různých místech.
- Sběr informací (tlaku, teploty, odběru energie) ze zařízení ve výrobní hale.

Při použití tohoto systému ve výrobní hale by bylo umožněno sbírat data o teplotě prvků strojů nebo prostředí, spotřebě energie, tlaku, vlhkosti vzduchu, pozici v prostoru u pohybujících se objektů a podobně. Tato data je užitečné ukládat do nějaké databáze pro případný monitoring a následné zpracování. Díky tomuto monitoringu lze odhalit potenciální problémy se zařízeními ještě než nastanou. Pokud třeba začne nějaké zařízení spotřebovávat víc energie než obvykle nebo se u něj začne zvyšovat tlak nad standardní hodnoty, pak může být díky monitorování tento problém včas odhalen a vyřešen. Díky tomu lze předejít poruchám, jejichž oprava by mohla být finančně velmi náročná, například preventivní výměnou opotřebované části stroje.

Mnoho firem dnes používá vlastní implementaci nějakého sběru dat. Tyto implementace však nejsou vždy optimální, škálovatelné, dobře zabezpečené a průběžně udržované. Z tohoto důvodu bylo rozhodnuto vytvořit bezpečné robustní centralizované řešení, které budou moci tyto firmy využít.

Bakalářská práce je rozdělena do 5 kapitol, kdy po úvodním představení problému uvedeme v Kapitole 2 teorii, kterou jsme využívali, v Kapitole 3 představíme IoT Cloud. Hlavní část (realizace) je popsána v Kapitole 4 a v Kapitole 5 je shrnuta celá práce a její výsledky.

1.1 Motivace

IoT technologie je téma, které je v dnešní době velmi aktuální. Sběr dat je nedílnou součástí pokroku a možnost shromažďovat data rychle a bezpečně pomocí internetových technologií je velice zajímavá myšlenka. Je nepraktické, aby si každý vytvářel svou vlastní infrastrukturu pro sběr dat. Praktičtější je vytvořit robustní a spolehlivou možnost, jak ukládat a zobrazovat data ve velkém. Proto došlo k rozhodnutí vytvořit část této infrastruktury, konkrétně worker, který se bude starat o komunikaci mezi klientem a IoT databází. Za klienta lze v tomto případě považovat jakékoliv IoT zařízení. Výsledný worker by měl být použitý hlavně pro průmyslové účely, kdy by se pomocí tohoto workeru ukládala data například z tovární haly. Výsledný systém nemusí být omezený samozřejmě jen na toto průmyslové použití. Například je možné tento systém nasazovat v rámci chytré domácnosti nebo pro sledování vývoje teploty v různých místnostech budovy. Stále je třeba počítat s tím, že se pracuje s citlivými daty. Je proto nutné během vývoje klást velký důraz na bezpečnost a spolehlivost každé části systému.

Svět je plný různých čidel, měřáků a jiných zařízení, která mohou být připojena k internetu. V posledních letech se klade velký důraz na Industry 4.0 (průmysl 4.0), kde je generováno velké množství dat, se kterými je třeba dále pracovat. Jako příklad si můžeme představit různé multiagentní systémy, například síť plně autonomních aut. Tato auta neustále získávají obrovská množství dat z okolí, která je zapotřebí někam ukládat. Možností využití IoT technologií je ohromné a je v nich velká budoucnost. Tento worker v kombinaci s IoT Cloudem firmy REX Controls s.r.o. by měl být použitelný pro širokou škálu podobných aplikací.

1.2 Příprava řešení

V rámci této práce se pokoušíme vytvořit program, který pomocí standardních protokolů umožní jednoduché připojení klienta a ukládání jeho dat do databáze. Tento program by měl být dostatečně univerzální a automaticky konfigurovatelný.

Příprava řešení probíhala v následujících krocích:

- Nejprve bylo nutné nastudovat, jaké jsou možnosti IoT technologií, konkrétně protokolu MQTT, který jsme vybrali z důvodu jeho velkého rozšíření a dobré dokumentace. Informace o tomto komunikačním protokolu jsou k dispozici na internetových stránkách [1].
- Následně bylo nutné rozhodnout, v jakém programovacím jazyce bude program realizován. Rozhodli jsme se pro implementaci v jazyce Python, protože s ním máme největší zkušenosti a je uživatelsky přívětivý. Přestože v jazyce C by program běžel rychleji, pro naše účely je jazyk Python dostačující. Jazyk Python disponuje řadou zajímavých knihoven souvisejících s naším problémem. Mezi ně patří například knihovny: json, time, requests, subprocess, logging, threading, os, signal, paho-mqtt, psycopg2. Zejména důležitá je pro nás knihovna paho-mqtt, která zajišťuje implementaci MQTT protokolu, dále knihovna psycopg2, která slouží k vyhledávání a ukládání do postgreSQL databáze. U všech těchto knihoven bylo nejprve nutné se detailně seznámit s jejich dokumentací.
- Dále bylo nutné vybrat, jaká implementace brokeru bude použita. Byl vybrán Mosquitto message broker, který implementuje MQTT protokol. Mosquitto byl zvolen z důvodu svého velkého rozšíření a spolehlivosti, navíc jsme s tímto brokerem měli už nějaké zkušenosti.
- Dalším krokem bylo seznámit se s IoT Cloudem firmy Rex Controls s.r.o., s kterým má tento worker spolupracovat. Bylo nutné zjistit, jakým způsobem bude worker s cloudem komunikovat. Současně bylo nutné seznámit se s architekturou tohoto cloudu.

[1] https://mqtt.org/

- Po seznámení se se všemi nástroji z předchozích kroků bylo dalším krokem otestování komunikace mezi cloudem a workerem na jednoduchém příkladu. Během celé přípravy řešení bylo myšleno na potřeby aplikací, pro které má systém sloužit.
- Následně byly navrženy jednoduché experimenty s MQTT brokerem a Cloudem. Cílem těchto experimentů bylo získání informací potřebných pro návrh architektury.
- Nakonec bylo potřeba vymyslet architekturu workeru a načrtnout diagram, podle kterého bude celý worker sestaven a zaintegrován mezi klienta a databázi.

Kapitola 2

Teorie

V této kapitole bude shrnuta teorie k použitým pojmům a technologiím, například IoT, MQTT, JSON-RPC, HTTP REST, postgreSQL, timescaleDB. Všem těmto technologiím bylo nutné porozumět, aby mohly být využity pro vytvoření finálního produktu.

2.1 IoT

IoT (internet of things) je síť fyzických zařízení, která jsou vybavena technologiemi pro získávání informací z reálného světa nebo interakci s reálným světem. IoT se běžně využívá například v průmyslu, chytrých domácnostech nebo v lékařství. Tato zařízení jsou propojena privátní nebo veřejnou sítí a každé z těchto zařízení je možno ovládat na dálku, aby plnilo požadovanou funkci. Informace mezi zařízeními je potom sdílena přes síť pomocí standardizovaných komunikačních protokolů. Tato „chytrá“ zařízení mohou mít různou podobu a velikost, od zařízení, která běžně nosíme, například chytré hodinky, po velké stroje. Každé takové zařízení obsahuje senzorový čip.

Jako příklad můžeme uvést chytré boty Lenovo, které obsahují čip, pomocí kterého získávají a analyzují sportovní data. Podobně lze mluvit o elektrospotřebičích, například ledničce nebo pračce, i tato zařízení lze ovládat pomocí internetových technologií. Dalším příkladem jsou bezpečnostní kamery, ke kterým lze získat přístup odkudkoliv ze světa. Tento seznam je velice rozsáhlý, vypsat všechny možnosti by bylo téměř nemožné.

Kromě soukromého využití lze IoT využít i ve veřejných službách. Různá zařízení, která mohou monitorovat vývoj počasí, poskytovat možnost sledování letadel, monitorovat operace v nemocnici nebo například čipování domácích zvířat. Data z těchto zařízení je možné sbírat v reálném čase, což vede ke zlepšení celého systému.

Díky zmenšování hardwaru a zvyšovaní jeho výkonu se IoT technologie rozšiřují do běžného života a rozsah využití stále roste. S jejich větším rozšířením roste samozřejmě důraz na bezpečnost. Problémy v kontextu s IoT se začaly objevovat v senzorových bezdrátových sítích nebo M2M (machine to machine) komunikaci. Celá architektura IoT musí být tedy dobře zabezpečena, aby nedošlo k problémům se soukromím [6].

2.2 Hrozby IoT

Při použití IoT technologií je třeba velmi dbát na bezpečnost. Tím, že jsou data posílána přes internet, je nutné je zabezpečit, protože mohou obsahovat citlivé nebo soukromé informace. S velkým nárůstem používání IoT technologií roste i potenciální nebezpečí. Důležitou součástí zabezpečení je autentizace a šifrování. Díky omezenému výpočetnímu výkonu a spotřebě energie (mnoho z těchto zařízení je napájeno bateriemi) je obtížné využívat standardních způsobů ochrany, jako je například detekce vniknutí nebo antivirus. Díky mnohdy primitivním ochranným mechanismům není těžké do takovýchto zařízení instalovat backdoor. Backdoor (zadní vrátka) je název metody, která umožňuje obejít standardní zabezpečení systému.

IoT zařízení jsou schopna monitorovat velké množství osobních informací, např. srdeční tep nebo teplotu v domě. Tato data jsou velice citlivá. Díky jejich ohromnému množství může jejich únik prozradit zdravotní

stav uživatele, jeho polohu nebo životní návyky (jak často sportuje, kde se pohybuje a podobně). Všechny tyto informace mohou být velmi snadno zneužity proti danému uživateli. Téma bezpečnosti v IoT technologiích je velice důležité a je třeba ho nebrat na lehkou váhu[14].

2.3 Edge computing

Většina aplikací dnes využívá cloud computing. Nicméně se čím dál více začíná používat i edge computing. Edge computing přesouvá zpracování dat blíže k jejich zdroji, což je kritické pro real-time aplikace, například řízení autonomních aut a podobně. Jeho základy je možné vysledovat až do roku 1960, i přesto je dnes považováno za novinku. Hlavní výhodou je menší náročnost na provoz sítě. Data se zpracují již na místě a přes síť jdou tedy jen ta nejnutnější. Tímto se zároveň zlepšuje i bezpečnost. Čím menší objem dat, tím větší bezpečnost. Zároveň ale dochází k přesunu nebezpečí přímo na zařízení. Díky tomuto principu je edge computing efektivnější a lépe škálovatelný. Existují i predikce, že edge computing nahradí cloud computing. Neznamená to ale, že cloud computing zanikne. Stále bude potřeba a bude se rozrůstat v návaznosti na edge computing. K navrácení k edge computingu dochází díky novým technologiím, jako například machine learning a podobně. [5]

2.4 Cloud computing

Cloud computing slouží k přesouvání služeb výpočtů nebo dat z koncových zařízení do centralizovaného zařízení (mimo pracoviště). Díky tomu lze k datům přistupovat z cloudu nebo z vnější sítě, což ulehčuje práci a mnohdy i šetří peníze. Další výhodou je snazší spolupráce napříč firmami. Obecně existují 3 typy Cloud computingu: Software as a Service (SaaS), Infrastructure as a Service (IaaS), Platform as a Service (PaaS). Jako další typ se někdy uvádí Storage as a Service(StaaS) [3, 11].

Jsou definovány 4 modely nasazení cloud služeb: veřejné cloudy, privátní cloudy, komunitní cloudy a hybridní cloudy. Každý tento model má své výhody a nevýhody, proto je důležité, aby si uživatel vybral model, který mu nejvíce vyhovuje. Veřejný model využívají uživatelé, kteří si pronajímají nějaký výpočetní výkon, nejčastěji k testování, sdílení dat nebo například na mailové služby. Privátní cloudy jsou vždy využívány pouze jednou společností, většinou se o provoz těchto cloudů stará společnost sama. Privátní cloudy jsou dražší, protože do nich firma musí zainvestovat a starat se o ně, na druhou stranu umožňují lepší možnosti zabezpečení. Hybridní cloudy jsou propojením privátního a veřejného cloudu. Pokud nestačí výpočetní výkon privátního cloudu, může uživatel doplnit chybějící výkon z cloudu veřejného. Model komunitního cloudu slouží pro vytvoření cloudu v uzavřené komunitě, například univerzitní cloud. K tomuto cloudu mají zpravidla přístup pouze členové dané komunity.

Cloud services zavádějí různé typy zákazníků. Konečný zákazník využívá vrstvy SaaS pomocí webového prohlížeče a získává potřebná data z IaaS vrstvy. Zákazníkovi podnikateli, který má přístup do všech vrstev, je umožněn i přístup do IaaS vrstvy, kde může vytvářet své vlastní aplikace. Poslední je vývojář, který může vylepšovat své aplikace pomocí SaaS vrstvy. [1]

2.4.1 Software as a Service

Tento typ služby je nejrozšířenější. Zákazník dostane hotový software, ale zařízení, na kterém software běží, si musí zařídit sám. Konkretní příklady jsou Drop Box, G suite, Microsoft office 365. [3]

2.4.2 Infrastructure as a Service

Tento typ služby poskytuje infrastrukturu, která je nutná pro správný běh softwaru. Využívá se v případě, že se poskytovatelé SaaS služeb nechtějí o infrastrukturu starat sami. Jedná se o servery, ke kterým je většinou přístup udělován virtualizací, aby se zvýšila efektivita a snížily náklady. Kromě software je poskytován i potřebný hardware.[3]

2.4.3 Platform as a Service

Tento typ služby slouží jako webové prostředí, které umožňuje vývojářům vytvářet cloudové aplikace. Uživatel se zde nemusí starat už téměř o nic Provider se postará o instalaci operačního systému a všeho nutného pro běh softwaru.[3]

2.4.4 Kritéria

Velký důraz se klade na bezpečnost neboli schopnost ochránit data dané organizace tak, aby nemohlo dojít k jejich zneužití. Dalšími kritérii jsou výkon, přístupnost a použitelnost (schopnost vyhovět požadavkům zákazníka), škálovatelnost a přizpůsobivost (možnost přizpůsobit se zákazníkovým požadavkům). [1]

2.5 Cloudové úložiště

Cloudové úložiště umožňuje ukládání dat do vzdáleného databázového systému, o který se stará třetí strana. Namísto standardního ukládání dat na pevné disky jsou data přes internet uložena do vzdálené databáze. Existuje mnoho různých systémů na ukládání dat na cloudu. Některé jsou velmi specifické, například ukládání fotek, emailů nebo zpráv. Jiné umožňují ukládání nezávisle na typu dat. Zařízení, na kterém běží cloudové úložiště, se nazývá data centrum.

Cloudové úložiště potřebuje k běhu jen data centrum, které je připojené k internetu. Uživatel, který chce uložit nějaká data, pošle kopie těchto dat do data centra. V případě že je chce uživatel získat zpět, připojí se na data server pomocí webového rozhraní. Server následně buď pošle data zpět, nebo umožní uživateli manipulaci s těmito daty přímo na serveru.

2.5.1 Storage as a Service

Tento typ umožňuje uživatelům ukládat data na vzdálená zařízení a přistupovat k nim odkudkoliv. Cloud storage systémy musí splňovat přísné požadavky na bezpečnost, spolehlivost a konzistentnost. [11]

2.6 PostgreSQL

PostgreSQL je relační databáze, která je vyvíjena již od roku 1986. Jedná se o open source databázi, kterou lze získat zcela zdarma. V dnešní době je postgreSQL jedna z nejlepších databází na trhu. Je tomu tak díky jejím specifickým vlastnostem.

- PostgreSQL je objektově relační typ databáze. Každá tabulka v postgreSQL definuje třídu. Mezi těmito třídami jsou nadefinované dědičnosti.
- Je standardizovaná, podporuje syntax SQL92 a z velké části SQL99.
- Jedná se o open source databázi.
- Umožňuje vytváření vlastních datových typů a funkcí.
- Podporuje vývoj klientských aplikací.
- Klade velký důraz na bezpečnost.

Fakt, že je postgreSQL rozšiřovatelná, umožňuje velkou míru přizpůsobení. Pokud chybí nějaká funkcionalita, může si ji uživatel dopsat nebo zjistit, zda už ji někdo nenapsal a nezveřejnil na internetu. [4, 13]

2.7 TimescaleDB

TimescaleDB je takzvaná time-series databáze. Jedná se o databázi, která ukládá časové řady. Každý řádek je doplněn o časový údaj a obsahuje informaci o měřených veličinách v daném čase. Tyto databáze ukládají velká množství dat lineárně (v reálném čase). TimescaleDB je rozšířením postgreSQL, sdílí s ní mnoho vlastností a využívá stejných nástrojů. Díky tomu se s ní pracuje v podstatě stejně jako se standardní verzí postgreSQL. Zároveň ale nabízí mnohá vylepšení oproti postgreSQL. Jednou z hlavních výhod je větší škálovatelnost. Při větším objemu dat, řádově 100 milionů řádků, dojde v případě postgreSQL k velkému zpomalení při insertu, timescaleDB si oproti tomu drží konstantní hodnotu.

Při benchmarku došlo k naměření následujících hodnot: při jedné miliardě řádků dokáže postgreSQL vkládat řádky rychlostí 5 tisíc za vteřinu, což zní jako dobré číslo, naproti tomu však TimescaleDB udržuje konstantní rychlost insertu kolem 110 tisíc řádků za vteřinu. Uložení miliardy dat tak trvalo postgreSQL 40 hodin, oproti tomu TimescaleDB to zvládla za pouhé 3 hodiny, a to už je rozdíl, který nelze přehlédnout. Lepších výsledků dosahuje TimescaleDB i při zpracování dotazů. Díky těmto vlastnostem se jedná o jasnou volbu v případě ukládání časových řad.[7]

2.8 HTTP REST

REST (Representational State Transfer) je komunikační rozhraní, které slouží ke komunikaci se serverem. Základním principem je abstrakce, objekty serveru jsou brány jako takzvané zdroje (resources). Jakákoliv informace, která může být pojmenována, může být zdrojem - dokument, obrázek nebo skupina jiných zdrojů. Každý zdroj je obdařen unikátním identifikátorem. Stav každého zdroje je nazýván "reprezentace zdroje". Tato reprezentace obsahuje data, metadata a link, díky kterému může klient pokročit do dalšího stavu. REST využívá bezstavového (neuchovává si nic z předchozí relace) client/server protokolu, nejčastěji HTTP s využitím metod POST, DELETE, PUT a GET. [9]

2.9 JSON-RPC

JSON-RPC (JSON remote procedure call) je jednoduchý transportní protokol, který využívá JSON (RFC 4627) pro kódování zpráv. Většinou se používá spolu s HTTP nebo websockets. Jeho největší výhodou je jednoduchost. RPC požadavek je vyřízen pomocí zaslání objektu na server a skládá se z několika částí: [12]

- JSONRPC - Řetězec, který specifikuje verzi JSON-RPC protokolu. Je nutné, aby byl exaktní, např. "2.0".
- Method - Řetězec obsahující jméno metody, kterou chceme volat (existují interní metody, které mají rezervována svá jména).
- Params - Strukturovaný soubor hodnot, které mají být použity jako parametry dané metody. Params nemusí být nadefinováno.
- ID - Identifikátor nastavený klientem, který by měl obsahovat string, číslo, případně null. Pokud ID není nadefinováno, automaticky se předpokládá, že se jedná o notifikaci. Server by měl odpovědět se stejným ID.

Pokud dojde k volání RPC, server reaguje odpovědí. Tato odpověď je opět ve formátu JSON a má následující části:

- JSONRPC - Řetězec, který specifikuje verzi JSON-RPC protokolu. Je nutné, aby byl exaktní, např. "2.0".
- Result - Tato část je v případě úspěšného požadavku vyžadována, naopak při chybě by neměla být přítomna. Hodnota této části závisí na metodě, která byla přijata na server.
- Error - Tato část je vyžadována pokud došlo k nějaké chybě. Pokud k chybě nedošlo, neměla by tato část existovat. Jedná se o objekt, který bude popsán níže.

- ID - Identifikátor, který musí být totožný s identifikátorem použitým při požadavku. Dojde-li k chybě při načtení tohoto ID, je vrácena hodnota NULL.

V odpovědi musí být vždy buď result nebo error, ale nikdy ne obě části. V případě, že nastane jakákoli chyba, odpověď serveru obsahuje objekt error. Součástí tohoto objektu jsou:

- Code - Číslo ve formátu integer, které reprezentuje zaznamenanou chybu.
- Message - Řetězec obsahující stručný popis chyby. Většinou jde jen o jednu větu.
- Data - Jednoduchá struktura dat, která obsahuje případné další informace o nastalé chybě. Tato část není vyžadována.

Příklady chyb:

- Code: -32700, Message: Parse error, vysvětlení: Na server byl přijat neplatný JSON, došlo k chybě při jeho parsování.
- Code: -32602, Message: invalid params, vysvětlení: Neplatné parametry volané metody.
- Code:- 32603, Message: internal error, vysvětlení: interní chyba JSON-RPC.

Příklad našeho požadavku:

```
{
  "id": 1,
  "method": "Metoda",
  "params": {
    "clientApiKey": "AAAA",
    "clientSecretKey": "BBBB"
  }
}
```

Příklad odpovědi:

```
{
    "result": [
        {
            "id": 1,
            "name": "name",
            "type": 2,
            "port": 0000,
            "settings": {
                "singleCredentialsPassword": "AAAA",
                "singleCredentialsUsername": "username"
            },
            "updatedAt": "2021-05-14T14:34:53+02:00",
            "instance": {
                "identifier": "AAAA",
                "name": "name",
                "database": "AAAA",
                "updatedAt": "2021-04-29T10:40:04+02:00"
            },
            "hostRole": {
                "identifier": "AAAA",
                "password": "AAA",
                "updatedAt": "2021-04-29T11:06:29+02:00"
            }
        }
    ],
    "error": null,
    "id": 1
}
```

2.10 MQTT

MQTT (MQ Telemetry Transport) je jednoduchý, volně přístupný protokol, který byl vytvořen za účelem vytváření klientských sítí s omezeným počtem vstupů. Tyto sítě jsou schopny jednoduše distribuovat různé druhy informací a dat v prostředí s malou šířkou pásma. MQTT protokol je založen na komunikaci pomocí systému publish/subscribe, který se používá v M2M komunikaci. Byl vytvořen jako nenáročný protokol z důvodu limitace CPU a šířky pásma, proto je vhodný k použití na zařízeních, která jsou limitována malým výkonem hardwaru a v oblastech, kde je špatné připojení k internetu a nestabilní odezva. Používá se hlavně v průmyslu, ať už v automobilovém, energetickém nebo telekomunikačním. I přesto, že byl tento protokol vytvořen pro dozorovou kontrolu a pro sběr dat (SCADA - supervisory control and data acquisition) v ropném plynárenském průmyslu, stal se velice populárním u široké veřejnosti pro použití v chytré domácnosti a dnes je jedním z nejvíce využívaných open-source protokolů. [2]

2.10.1 Jak funje MQTT

Ve snaze o co nejlepší využití omezené šířky pásma nahrazuje model publish/subscribe tradiční architekturu client/server, ve které dochází ke komunikaci přímo s endpoitem (koncový uzel sítě, sloužící jako cíl dané komunikace). V publish/subscribe modelu je klient, který posílá zprávu (publisher), oddělen od klienta, který zprávu přijímá (subscriber). Protože nedochází k přímé komunikaci mezi publisherem a subscriberem, o spojení se musí starat třetí strana neboli broker. MQTT klient může být ve stavu subscriber, publisher, anebo může zároveň zprávy přijímat i odesílat (publisher i subscriber). V tomto modelu se může více klientů přihlásit k jednomu brokeru a požádat o subscribe jednoho topicu (tématu). Pokud na tento topic nějaký publisher pošle zprávu, všichni klienti, kteří mají stav subscribe pro tento topic, dostanou danou zprávu. Pokud se spojení mezi subscriberem přeruší, broker automaticky začne zprávy ukládat do bufferu. V momentě, kdy se daný subscriber znovu připojí, broker mu všechny zprávy odešle. Dojde-li k nečekanému odpojení klienta ve stavu publisher od brokeru, může broker ukončit spojení a všem klientům, kteří mají přihlášený subscribe, rozešle přednastavenou zprávu s instrukcemi od publishera.

Pro shrnutí: publisher posílá zprávy, subscriber přijímá zprávy z topicu, o které má zájem. Broker předává zprávy mezi subscriberem a publisherem. Publisher a subscriber jsou MQTT klienti. Tito klienti komunikují pouze s brokerem. Klientem může být jakékoliv zařízení nebo aplikace, např. mikrokontroléry, malé počítače (arduina) a PLC, skripty na osobních počítačích nebo servery. [2]

2.10.2 MQTT broker

MQTT broker je „most“ spojující klienty, kteří zprávy posílají, s klienty, kteří na zprávy čekají. Všechny zprávy musí projít přes broker. Můžeme si ho představit jako poštu. Pro úspěšné doručení balíčku musí nejprve na poštu a až poté z pošty do místa určení. Broker by měl být schopný obsloužit miliony připojených MQTT klientů. Z toho důvodu je dobré při vybírání dbát na to, aby umožňoval škálovatelnost, integrovatelnost a nebyl náchylný k poruchám. [2]

2.10.3 Typy MQTT zpráv

MQTT relace je složena ze čtyř částí: připojení, autentifikace, komunikace a ukončení (terminace). Relaci započne klient vytvořením spojení s brokerem pomocí TCP/IP protokolu s použitím standardního portu anebo libovolného přednastaveného portu, který je brokerem nadefinovaný. Pokud broker rozpozná starého klienta, dojde k obnovení starého spojení (pouze pokud zpráva CONNECT neobsahuje příznak clean session). MQTT standardně používá port 1883 pro nešifrovanou komunikaci a port 8883 pro šifrovanou, kde využívá SSL/TLS (Secure Sockets Layer/Transport Layer Security). Během handshaku dojde k předání certifikátů, které jsou použity pro autentifikaci. Použití SSL/TLS nemusí být vždy možné, protože MQTT je primárně používáno pro IoT, kde je omezený výkon zařízení a připojení k internetu, které v některých případech není ani vyžadováno. V těchto případech je autentifikace řešena pomocí cleartext jména a hesla, která jsou odeslána klientem na server v rámci zprávy CONNECT. V případě navázání spojení odešle zařízení zprávu CONNECT. Zpráva velmi často obsahuje příznak clean session, což zajistí, že zařízení nebude přihlášeno k odběru žádného tématu. Broker na tuto zprávu odpovídá zprávou CONNACK, kterou potvrzuje spojení. Některé brokery umožňují přístup anonymním klientům, v tomto případě není heslo a jméno uvedeno [2, 8].

2.11 Porovnání OPC UA a MQTT

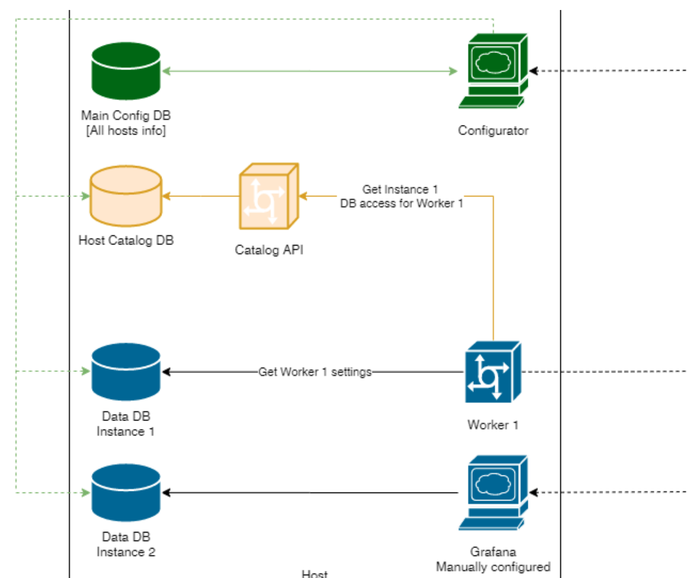
V této sekci budou stručně porovnány komunikační protokoly OPC UA (Open Platform Communications – Unified Architecture) a MQTT. Jak už bylo zmíněno, rozhodli jsme se využít MQTT pro jeho jednoduchost a velké rozšíření. Jednoduchost ovšem má i své nevýhody, a v tento moment je třeba představit OPC UA. Protokol OPC UA je standardizovaná průmyslová komunikace, která je mnohem jednotnější, bezpečnější a propracovanější než MQTT. Proto je v průmyslu používán spíše tento protokol. Nicméně velká složitost OPC UA nevyhnutelně vede k větší náročnosti na implementaci. Komunikace přes tento protokol musí obsahovat mechanismy, které zajišťují bezpečný přenos dat. Je využíváno šifrování, použití přihlašovacích údajů a správa oprávnění. MQTT je oproti tomu velmi jednoduchý a spíše se využívá v prostředích, kde není kladen tak velký důraz na standardizaci. Zároveň ale MQTT poskytuje velkou výhodu v jeho nenáročnosti na internetové připojení. Tyto dva protokoly jsou každý úplně jiný a každý má své uplatnění. OPC UA je komplexnější, propracovanější a zajišťuje mnoho věcí, které MQTT neobsahuje. Zajímavostí je, že OPC UA může v určité konfiguraci používat MQTT jako jeden ze svých interních transferních protokolů.

Kapitola 3

IoT Cloud

IoT Cloud firmy REX Controls s.r.o. je cloud vytvořený pro ukládání časových řad. K tomuto účelu interně využívá postgreSQL databázi s rozšířením timescaleDB (viz sekce 2.7). Tento Cloud je aktuálně v poslední fázi vývoje a bude v budoucnu využíván pro komerční použití. Jako příklad použití může sloužit tovární hala, kde může být monitorována teplota těžkých strojů, tlak lisu, spotřeba energie daného stroje nebo vibrace. Tyto informace mohou být periodicky ukládány na cloud a mohou být jednou za určité období vyhodnoceny a využity pro lepší optimalizaci celé haly. Nebo lze pomocí těchto informací odhalit součástky, které už dosluhují a nahradit je, ještě než dojde k většímu poškození. Dále lze monitorovat spotřebu energie a vyhodnotit, zda se provoz daného stroje vyplatí nebo zjistit, že v danou hodinu lze připojit nabíjení vysokozdvižných vozíků, a podobně.

3.1 Popis IoT cloudu firmy REX controls s.r.o



Obrázek 3.1: Toto schéma odpovídá situaci, kdy se konfigurátor i databáze nacházejí na jednom serveru (localhost).

Tento Cloud se skládá z několika částí. Základem Cloudu je webová aplikace, která je na Obrázku 3.1 zobrazená jako konfigurátor. Tento konfigurátor má u sebe vlastní databázi (Obrázek 3.2 Main Config DB), ve které jsou uloženy všechny věci týkající se konfigurátoru samotného. Mezi ně patří například přihlašovací

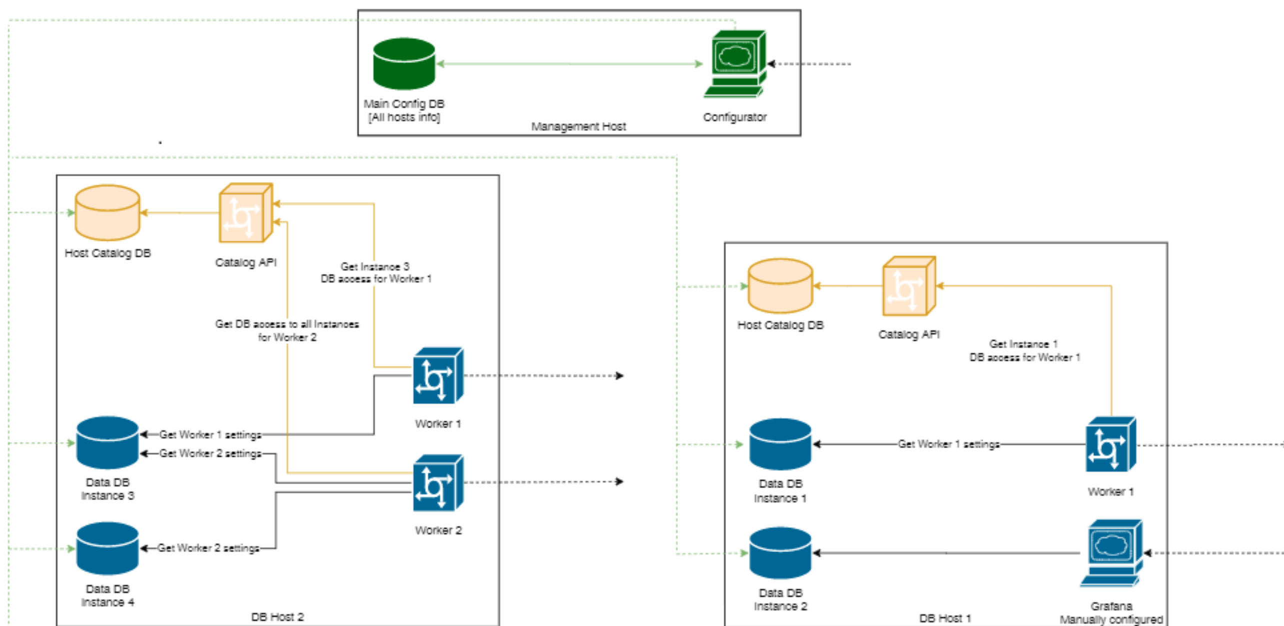
údaje uživatelů, různé tokeny a podobně. Dále se zde nacházejí informace o hostech, instancích, klientech, službách a rolích. Prvním krokem je nainstalování hostu (virtuální nebo fyzický server) s postgreSQL a vytvoření patřičného záznamu v konfigurátoru. Na obrázku se jedná o localhost, ale je možné vytvořit host na cizím stroji a připojit se přes IP adresu (viz Obrázek 3.2).

V momentě, kdy přijde nový zákazník a zažádá o kus Cloudu, dojde k vytvoření nové instance. Tato instance může reprezentovat firmu, oddělení nebo tým. Data na této instanci jsou oddělena od ostatních přístupovými právy. Instance lze ale rozdělit i fyzicky (různé části serveru nebo vlastní virtuální host pro každou instanci). Vytvořením instance dojde na daném hostu k vytvoření nové databáze (Data DB). S touto databází lze už pracovat například přes worker nebo například Grafanu (nástroj pro vizualizaci dat více na [1]). Tyto nástroje je ale nutné manuálně konfigurovat (manuálně nastavit přístupové údaje).

Cílem této práce bylo vytvořit worker tak, aby mimo jiné splňoval tyto body:

- Automatická konfigurace a reakce na změny v nastavení databáze.
- Bezpečnost (klient nemůže zasáhnout přímo databáze).

Aby byly tyto body splněny, worker využívá speciální databáze (na Obrázku 3.2 pojmenované jako Host Catalog DB). Do této databáze se zrcadlí potřebné informace z hlavní databáze (Obrázek 3.2 Main Config DB), ovšem už pouze informace potřebné pro daný host. K těmto informacím z důvodu bezpečnosti přistupuje worker pouze přes speciální API pro to vytvořené (na Obrázku 3.2 pojmenované Catalog API [2]). Toto API po připojení na databázi poskytne workeru všechny informace potřebné k jeho činnosti (např. jaké se zde nacházejí instance, na co má worker práva a pod jakou rolí se lze připojit).



Obrázek 3.2: Schéma cloudu s více hosty - V horní části se nachází konfigurátor na samostatném serveru, pod ním se nacházejí dva hosty, které tento konfigurátor nastavuje.

[1] `https://grafana.com`

[2] API je software, který funguje jako prostředník mezi dvěma aplikacemi. Obvykle si zažádá o nějaké informace na server a poté vrátí vše potřebné uživateli.

V případě více zákazníků, kde každý z nich vyžaduje vlastní server, vypadá schéma následovně (viz Obrázek 3.2). Na hlavním hostu běží už pouze konfigurátor a jeho databáze. Na tento konfigurátor se zákazníci připojují přes web, jinak je uzavřený. V této databázi jsou uloženy informace o obou hostech, včetně toho, jak se na ně připojit a jaké jsou přístupové údaje k jejich Host Catalog DB. Vytvořením instance na Hostu 1 dojde k zapsání této instance do Catalogu příslušnému Hostu 1 a vytvoření databáze. Dále vše funguje jako v předchozím případě. Worker zavolá Catalog API, získá potřebné údaje a připojí se na databázi, žádné další informace nepotřebuje. Na druhém hostu dojde k podobnému nastavení bez ohledu na to, kde se nachází. Na scénáři zobrazeném na Obrázku 3.2 si lze všimnout, že Worker 1 (například MQTT) z Catalogu dostane právo pouze na Instanci 3, zatímco Worker 2 dostane práva k oboum instancím. Výhodou tohoto přístupu je, že se databáze navzájem neovlivňují. V případě, že dojde k chybě na serveru s konfigurátorem, ostatní hosty nejsou ovlivněny a fungují normálně dál. Tyto hosty tedy fungují nezávisle a zcela odděleně. Další výhodou je jednoduchost změny všech přístupových údajů při jejich úniku, kdy následně každý worker zjistí, že došlo k chybě a automaticky si obstará údaje nové.

IoT Cloud firmy REX Controls s.r.o. se nachází ve finální fázi vývoje a z tohoto důvodu ještě neexistuje oficiální dokumentace. Ve finální verzi by měl zastupovat všechny typy služeb SaaS, IaaS. Pokud dojde k propojení s řídícím systémem REXYGEN, který přímo umožňuje komunikaci přes MQTT protokol, lze mluvit i o službě PaaS. Cloud má sice vlastní API, ale bylo potřeba vytvořit worker mezi MQTT klientem a databází, které bude obstarávat zabezpečení a zpracování zpráv. [10]

3.2 Catalog API

Jak už bylo řečeno, z důvodu dalšího zabezpečení worker nezasahuje přímo do databáze catalogu, ale využívá JSON-RPC API, které bylo pro tento účel vytvořeno. Právě toto API umožňuje automatickou konfiguraci celého workeru. Catalog API poskytuje metodu getServices, která vrátí všechny potřebné informace pro připojení k daným službám. Toto API je voláno pomocí HTTP metody POST na dané URL, více v kapitole 2.8. Data obsažená v tomto postu jsou v JSON-RPC struktuře, viz kapitola 2.9, a musí obsahovat id, jméno metody a parametry této metody. Tato data pak vypadají následovně:

```
data = {
    "id": 1,
    "method": "getServices",
    "params": {
        "clientApiKey": clientApiKey,
        "clientSecretKey": clientSecretKey
    }
}
```

Parametry clientApiKey a clientSecretKey jsou získány z konfiguračního souboru workeru umístěného na serveru.

Kapitola 4

Realizace vlastního komunikačního rozhraní MQTT

V této kapitole bude představeno námi realizované řešení workeru zajišťujícího komunikaci mezi databází a klientem. Konkrétně v kapitole 4.1 představíme návrh architektury, v kapitole 4.2 a 4.3 se budeme věnovat experimentu s Cloudem a MQTT, nakonec v kapitole 4.4 popíšeme jednotlivé části programu.

Jako první bylo nutné udělat si představu o tom, jak celý program bude vypadat a rozmyslet si, jaké nástroje budou použity. Pro implementaci jsme zvolili programovací jazyk Python. Jedná se o interpretovaný jazyk. Interpretované jazyky mají speciální program zvaný interpreter, který provádí zdrojový kód za běhu. Výhodou jazyka Python je jeho jednoduchost a velké rozšíření. Nevýhodou je, že není tak rychlý jako jiné jazyky. V našem případě je ale rychlost Pythonu zcela postačující.

Testování probíhalo vždy po určitém úseku nahráním a vyzkoušením dané části kódu na serveru pomocí Putty, což je program umožňující připojení na sever pomocí protokolu SSH [1].

4.1 Experiment s MQTT

Experiment s MQTT jsme prováděli, abychom otestovali možnosti tohoto protokolu. Pro provedení tohoto experimentu bylo nutné lokálně zprovoznit MQTT broker. Postupovali jsme v následujících krocích.

- Nainstalovali jsme Mosquitto broker a pomocí příkazové řádky ho spustili a otestovali jeho funkčnost.
- Připravili jsme jednoduchý publisher a subscriber ve formě Python skriptu. Pomocí těchto skriptů jsme otestovali odeslání a příjem zprávy.
- Dále jsme se pustili do testování různých možností Mosquitto brokeru pomocí upravování konfiguračního souboru. Možnosti tohoto brokeru jsou velice rozsáhlé, většinu jeho funkcí jsme vůbec nevyužili.

Při testování se nám podařilo změnit port, na kterém broker běží a definovat různé omezující parametry pro uživatele, jako například umožnit uživateli pouze zasílat zprávy nebo naopak pouze číst nebo omezit jeho přístup pouze na konkrétní témata (topic). Dále lze také uživateli nastavit přihlašovací jméno a heslo.

Během testování jsme narazili na problémy týkající se autentizace uživatele. Dle původního plánu mělo docházet k upravě uživatelských údajů dynamicky pomocí přidávání a odebírání uživatelů a jejich údajů (heslo a jméno) podle toho, kdo se zaregistruje. Bohužel v Mosquitto je pro tyto účely nutné přepsat konfigurační soubor a vyresetovat broker, což je pro naše účely velmi nepraktické. Nalezli jsme několik možností, z nichž většina byla pro naše účely zbytečně komplikovaná. Zkusili jsme tedy využít dostupné API, které umožňovalo automaticky shromažďovat a upravovat klienty v databázi a následně je přidávat nebo odebírat z Mosquitta. Toto API se zdálo jako přesně to, co bychom potřebovali. Bohužel toto API nebylo udržované a bylo tedy zastaralé. Proto by bylo nutné ho z velké části předělat. Z toho důvodu jsme tuto možnost zavrhli.

[1] https://www.putty.org

Nakonec jsme vymysleli jiný způsob řešení problému s autentizací. Toto řešení spočívá v tom, že každý uživatel bude autentizační data posílat ve zprávě spolu s daty, která chce uložit. Tuto zprávu pak zpracuje subscriber. Ten si z ní nejdříve vybere autentizační údaje, které využije k selectu z databáze, pomocí kterého zjistí, zda se jedná o validního uživatele. Tímto jsme obešli problém s dynamickou změnou uživatelů.

4.2 Experiment s Cloudem

V této podkapitole popíšeme experiment s Cloudem. Tento experiment provádíme, abychom zjistili, jak nastavit Cloud a jakým způsobem se k němu budeme připojovat. Experiment byl proveden v následujících krocích:

- Nejprve bylo třeba naučit se ve webovém rozhraní Cloudu, zaregistrovat uživatele a vytvořit danou tabulku. V prvním kroku byl vytvořen host, kterému byli přiděleni uživatelé.
- Na tomto hostu byla vytvořena instance, do které byl opět přidán náš uživatel a byla mu přidělena práva.
- Pomocí tohoto uživatele jsme na instanci vytvořili testovací projekt a v něm datovou tabulku.
- Dále bylo třeba založit servicu (službu), která poběží na nějakém portu. Tímto krokem byl Cloud již připraven.
- Nyní bylo třeba otestovat Catalog a jeho API. K tomuto účelu bylo využito HTTP REST a JSON-RPC. Pomocí jednoduchého requestu jsme otestovali, zda po zaslání POST requestu na Catalog dostaneme zpět požadované informace. Obdrželi jsme následující odpověď:

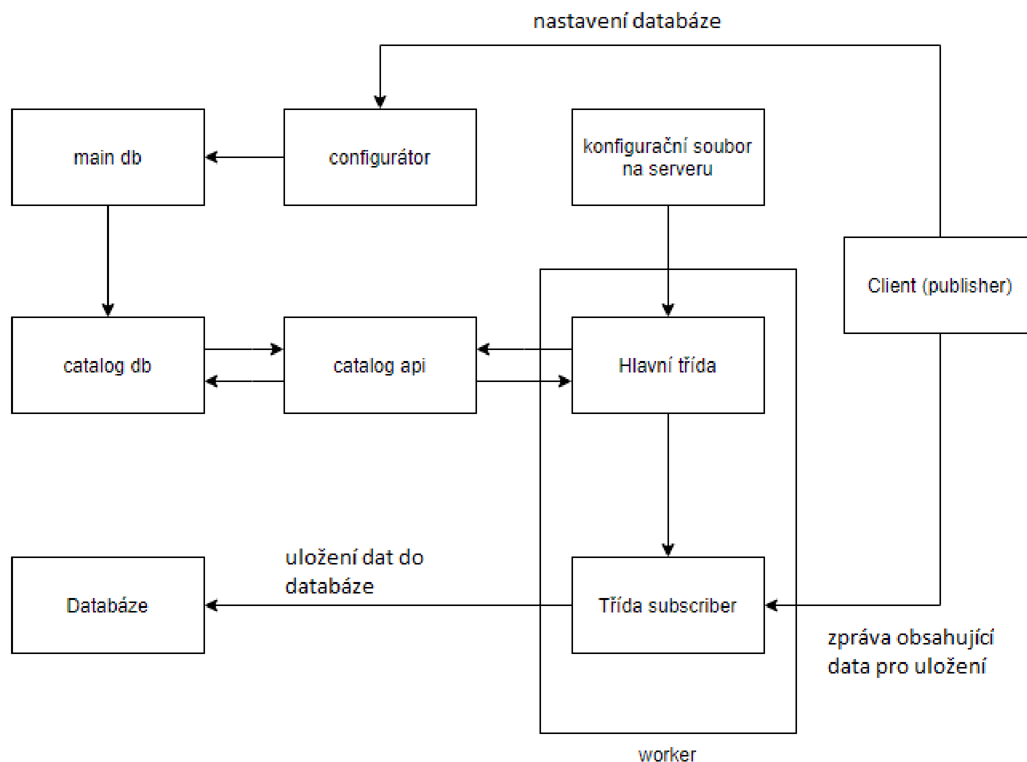

```
{
    "result": [
        {
            "id": 1,
            "name": "MQTT Service",
            "type": 2,
            "port": 8885,
            "settings": {
                "singleCredentialsPassword": "AAA",
                "singleCredentialsUsername": "AAAA"
            },
            "updatedAt": "2021-05-14T14:34:53+02:00",
            "instance": {
                "identifier": "mqtt_test",
                "name": "MQTT test",
                "database": "tsci_mqtt_test",
                "updatedAt": "2021-04-29T10:40:04+02:00"
            },
            "hostRole": {
                "identifier": "mqtt_service",
                "password": "AAA",
                "updatedAt": "2021-04-29T11:06:29+02:00"
            }
        }
    ],
    "error": null,
    "id": 1
}
```


Z této odpovědi lze vyčíst, že je zaregistrovaná pouze jedna service s příslušným ID 1 na portu 8885. Dále jsou zde informace potřebné k připojení k databázi, nastavení, kdy byla service updatovana, informace o hostRole (identifikátor, heslo a kdy byla provedena poslední změna). Jedná se o roli pro připojení do postgreSQL databáze.

Tyto informace jsou zpracovány a dále využity v hlavním programu.

4.3 Návrh architektury

Dalším krokem bylo vytvořit vhodnou architekturu, která by splňovala všechny naše požadavky.



Obrázek 4.1: Diagram celého systému - worker se skládá z hlavní třídy a třídy Subscriber. Je vidět, že klient nastavuje pouze konfigurátor a dále už komunikuje pouze se třídou subscriber.

Myšlenka je taková, že klient nastaví svojí databázi v konfigurátoru a tato informace se postupně dostane až do catalog databáze, kde si ji worker pomocí Catalog API získá, aktivuje MQTT broker v novém příkazovém řádku a na novém vlákně spustí třídu Subscriber. Klient už pak jen pomocí jednoduchého publishera posílá zprávy, které MQTT broker přeposílá workeru. Klient tedy s workerem komunikuje jen nepřímo. Všechny informace potřebné pro správné fungování worker získá z Catalog API.

Celý worker se skládá ze dvou tříd: hlavní třída a třída subscriber. Většina věcí se odehrává v rámci hlavní třídy. Tato třída zajišťuje správnou konfiguraci celého workeru, komunikuje s Catalog API a podle získaných informací aktivuje druhou třídu. Informace o připojení ke Catalog API načte z konfiguračního souboru, který je uložen na serveru. Druhá třída subscriber je spuštěna vždy na novém vlákně a běží paralelně. V rámci této třídy běží subscriber. Zde jsou zpracovány příchozí zprávy a pokud vše jde podle plánu, jsou i uloženy do databáze.

Využity byly tyto knihovny:

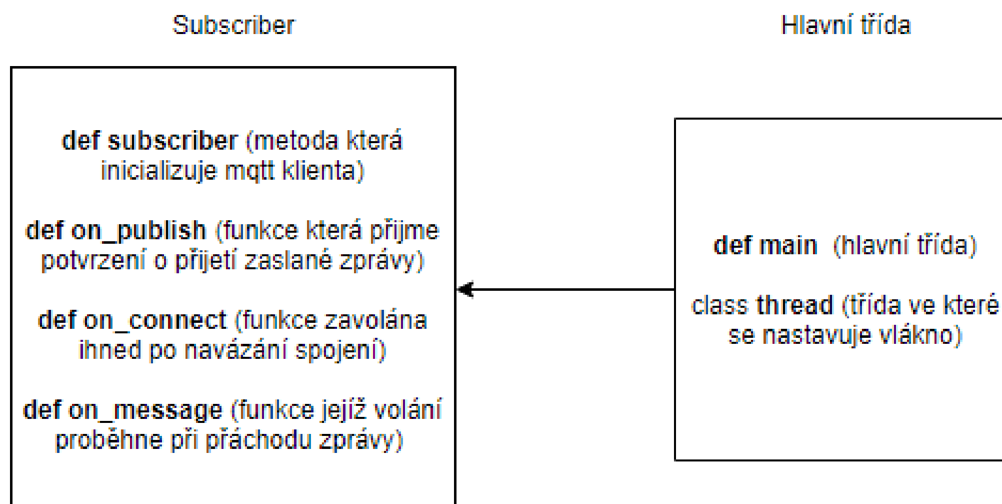
- Request: knihovna umožňující zasílání HTTP requestů.
- Subprocess: tato knihovna umožňuje spouštění nových procesů. V našem případě každý proces reprezentuje jeden broker[2].

[2]https://docs.python.org/3/library/subprocess.html

- Threading: knihovna, která umožňuje vytváření vláken a paralelizaci, tato knihovna byla důležitá pro současný běh několika subscriberů najednou[3].
- Psycopg2: knihovna umožňující práci s databází. V našem případě byla využita třída cursor a její metody execute a fetchone(), dále třída connection s jejími metodami connect a close.

4.4 Části programu

V této kapitole bude detailně popsána hlavní třída, která se stará o nastavení brokeru, a subscriber, který zajišťuje zpracování dat a uložení těchto dat do databáze. Každá z tříd má tak svoji specifickou roli a je třeba, aby spolu komunikovaly a zajistily tak stabilní přenos dat mezi klientem a databází. Pro stabilitu přenosu klademe za důležité zejména rychlé obnovení spojení mezi subscriberem a brokerem v případě výpadku spojení. Zprávy, které broker během tohoto výpadku dostane, jsou uloženy do dočasné paměti brokeru a po opětovném automatickém připojení jsou všechny tyto zprávy odeslány najednou v pořadí, v jakém byly přijaty brokerem.



Obrázek 4.2: Diagram obou tříd a příklady jejich metod. V případě vytvoření brokeru hlavní třída inicializuje třídu subscriber na novém vlákně.

4.4.1 Hlavní třída

Hlavní třída je základem celého workeru. Jejím úkolem je načíst data ze serveru a z konfiguračního souboru, zpracovat je a pomocí těchto dat nastavit Mosquitto broker a spustit subscriber. Zároveň se program stará o kontrolu nakonfigurovaných služeb a přístupových údajů. Při přidání nové služby je třeba založit nový broker a subscriber. Tento program byl vytvořen na základě předem promyšlené architektury, viz kapitola 4.1.

Po spuštění programu dojde k načtení konfiguračního souboru, který se nachází na serveru a jsou v něm v JSON formátu uložena následující data: clientApiKey, clientSecretKey, catalogUrl, databaseHost, mqttHost

[3] https://docs.python.org/3/library/threading.html

a savepath. Program tato data načte a rozdělí je do příslušných proměnných, načež začne opakovaně v hodinovém intervalu odesílat JSON-RPC request jako POST (pomocí knihovny request) na adresu catalogUrl s voláním metody getservices s parametry clientSecretKey a clientApiKey, které získal z konfiguračního souboru. Pokud API rozezná clientSecretKey a clientApiKey, odešle zpátky informace o všech službách, které jsou aktuálně zaregistrovány (z webového rozhraní). V tomto případě worker nezasahuje přímo do databáze, ale jen volá předem připravené API. Odpověď je opět v JSON formátu a obsahuje například: ID, name, port, updateAt, settings, informace o instanci a přihlašovacích údajích do databáze, viz kapitola 4.2.

Program v první iteraci vytvoří v Python slovník (struktura dat obsahující vždy klíč a jemu příslušné hodnoty) a přidá do něj data, která získal z Catalog API. Poté při každé iteraci projde slovník a porovná zda souhlasí informace ze slovníku s těmi, co získal pomocí Catalog API. Následně mohou nastat tři situace:

- V případě, že se ID které přišlo z Catalog API ve slovníku nenachází, vytvoří program konfigurační soubor pro Mosquitto broker, do kterého uloží port náležející danému ID. Následně spustí subprocess, ve kterém zinicializuje Mosquitto broker za pomocí předem vytvořeného konfiguračního souboru. V dalším kroku program spustí na novém vlákně funkci subscribe z třídy subscriber, které dodá číslo portu, na kterém právě spustil broker, a informace, které zjistil z requestu: settings, instance a hostRole. Všechny výše uvedené informace předá této funkci v parametrech. Dále přidá nové ID s jeho informacemi do slovníku.

- Pokud zjistí, že oproti předchozímu requestu došlo k odstranění některé služby (ve slovníku je ID, které ale nepřišlo z Catalog API), ukončí broker s tímto ID, smaže jeho konfigurační soubor a ukončí vlákno, na kterém běží subscriber pro danou službu. Nakonec odstraní dané ID ze slovníku.

- Data ve slovníku se rovnají datům získaným pomocí Catalog API. To znamená, že nedošlo ke změně a není tedy vyžadována žádná další akce.

Program také kontroluje, zda nedošlo ke změně proměnné updateAt. Pokud ano, znamená to, že byla služba změněna, a program se pokusí najít, k jaké změně došlo (například změna portu).

Velká pozornost byla kladena na paralelizaci daného programu. Při paralelizaci bylo třeba spustit nový proces (broker) a také nové vlákno obsahující třídu subscriber. K tomu bylo využito knihoven subprocess a threading viz kapitola 4.1. Důležité bylo ošetřit případ, kdy z jakéhokoliv důvodu dojde k odstranění instance (například při chybě chodu serveru nebo při odstranění klienta). V tom případě je třeba korektně ukončit běžící proces a vlákno, na kterém běží subscriber. Abychom ošetřili tento problém, potřebovali jsme zjistit ID procesu a vymyslet, jak ukončit vlákno z hlavního programu. Z toho důvodu bylo nutné si ukládat všechny informace o stávajících připojeních do slovníku s příslušným ID, aby mohl být daný proces identifikován a následně ukončen.

4.4.2 Subscriber

Třída subscriber zajišťuje, aby docházelo ke správnému příjímání a zpracovávání zpráv. V této třídě jsou implementovány následující metody a funkce:

- subscriber - hlavní metoda

- callback funkce on_publish, tuto funkci jsme nevyužili, protože je volána poté, co broker potvrdí, že byla zpráva přijata. To znamená že je využita pouze u publisheru.

- callback funkce on_message, tato funkce je zavolána pokaždé, když přijde nějaká zpráva. V rámci této funkce proběhne hlavní část našeho subscriber.

- callback funkce on_connect. K volání této funkce dojde po připojení k brokeru, jde většinou jen o potvrzení připojení.

Vykonávání třídy probíhá následujícím způsobem:

i) V momentě, kdy je spuštěna metoda subscriber, uloží všechny parametry důležité pro připojení k databázi do globálních proměnných. Poté nastaví klienta, k němu callback funkce a připojí se na broker na portu, který dostala v parametrech. Následně je spuštěna funkce loop_forever(), která tuto

metodu zacyklí a zajistí opětovné připojení k brokeru. Z tohoto důvodu se nic, co následuje po tomto zavolání, neprovede a tudíž toto vlákno nikdy neskončí. Díky tomu bylo nutné vymyslet způsob, jak toto vlákno v případě potřeby ukončit.

Jedna z možností byla poslat na broker zprávu, kterou subscriber vyhodnotí a vypne se. Tento způsob byl nakonec zavrhnout, protože by bylo třeba vytvořit další třídu publisher a zároveň by vznikalo bezpečnostní riziko, protože danou zprávu by mohl odeslat kdokoliv. Jinou variantou, která vede na spolehlivější řešení, bylo použití výjimky. V hlavním programu jsme schopni vyvolat výjimku, zachytit jí v třídě subscriber pomocí try catch bloku a celou třídu, a tím pádem i vlákno, na kterém běží, ukončit. Toto řešení se nakonec ukázalo jako vhodnější, a proto bylo využito v našem programu.

ii) Ve chvíli, kdy je obdržena nějaká zpráva, dojde ke spuštění metody on_message. Tato zpráva obsahuje následující informace: Apikey, Secretkey, Project, ObjectTableId, ObjectId, Data. Jako příklad uveďme následující zprávu ve formátu JSON.

```
msg = {
      "apiKey": "aaaa",
      "secretKey": "bbbb",
      "project": "project1",
      "objectTableId": "1",
      "objectId": "1",
      "data":
      {
        "time": "2021-05-22 15:00:00",
        "value": "0.5"
      }
}
```

Metoda on_message nejprve uloží přijatá data do příslušných proměnných a pokusí se připojit k databázi pomocí údajů, které jsou uloženy v globálních proměnných, které byly předány metodě subscriber z hlavního programu. Pokud připojení proběhne v pořádku, dojde k provedení několik SQL příkazů (selectů) na databázi, aby bylo možné ověřit uživatele a zjistit jméno a ID datové tabulky, do které mají být data uložena. Tyto selecty nejprve zajistí validitu uživatele a poté získají identifikační údaj datové tabulky.

- Pomocí tohoto selectu je ověřeno, že ApiKey a SecretKey, které byly přijaty ve zprávě, skutečně odpovídají některému z uživatelů a zároveň je zjištěno jeho individuální ID.
- V dalším selectu pomocí projektu a objectTableId, které byly součástí zprávy, zjistíme jméno objektové tabulky.
- Pomocí tohoto jména tabulky, objectId a jména projektu získáme ID datové tabulky.
- A pomocí tohoto ID a jména projektu získáme již jméno datové tabulky.

Teprve nyní máme všechny potřebné informace k tomu, abychom mohli zapsat daná data do datové tabulky. Pokud jakýkoliv z těchto selectů dopadne neočekávaně, k žádnému zápisu dat nedojde, abychom zabránili nepovolanému zápisu do databáze. V prvních selectech zároveň dochází ke kontrole, zda vůbec existuje projekt a tabulka, do které chceme zapisovat.

iii) Nyní dojde k insertu zaslaných dat do tabulky, jejíž jméno a ID jsme získali. Ihned po zápisu dat je spojení s databází ukončeno a metoda on_message končí. Při přijetí další zprávy se celý tento proces opakuje.

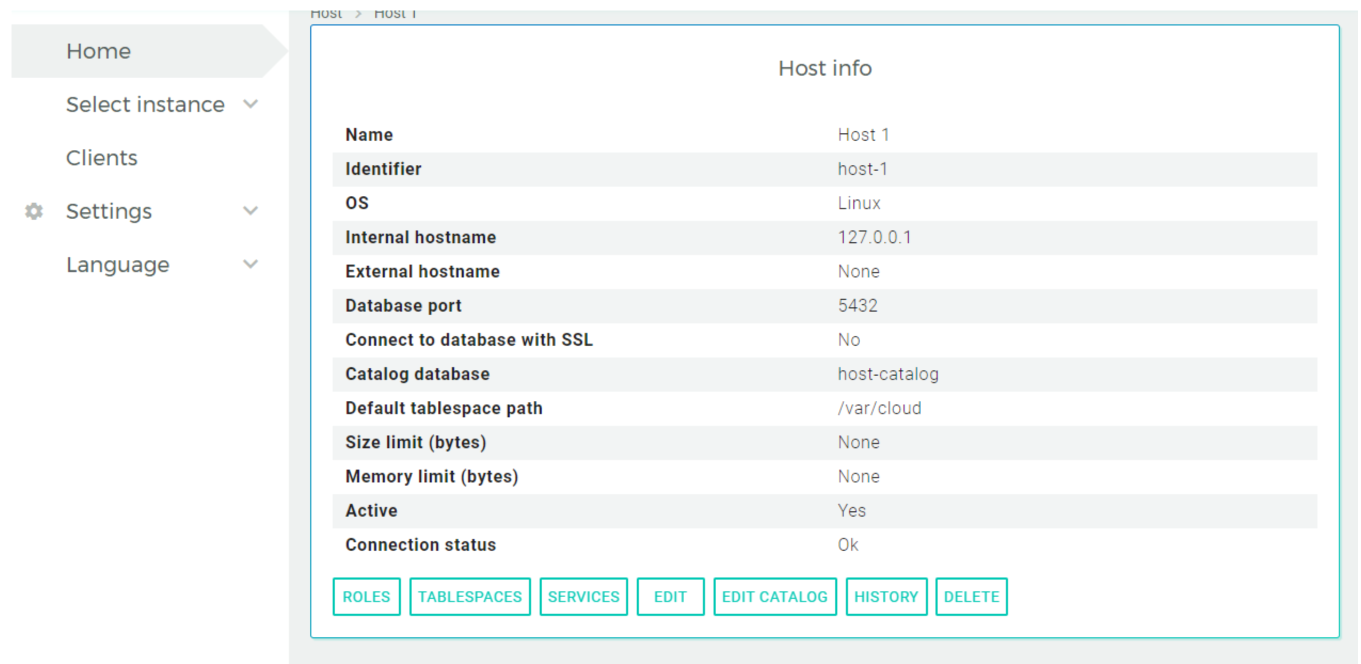
Kapitola 5

Výsledky

V této kapitole budou představeny výsledky experimentu, kdy byla nejprve vytvořena databáze na serveru, poté byl spuštěn worker a pomocí jednoduchého publisheru byla odeslána testovací data.

5.1 Experiment

Pro test celého postupu je nejdříve vytvořen host. Na tomto hostu běží PostgreSQL (viz kapitola 2.6). Každý host obsahuje jeden Catalog (pro možnosti automatické konfigurace) a jemu příslušné API. Příklad vytvořeného hostu je zobrazen na Obrázku 5.1.

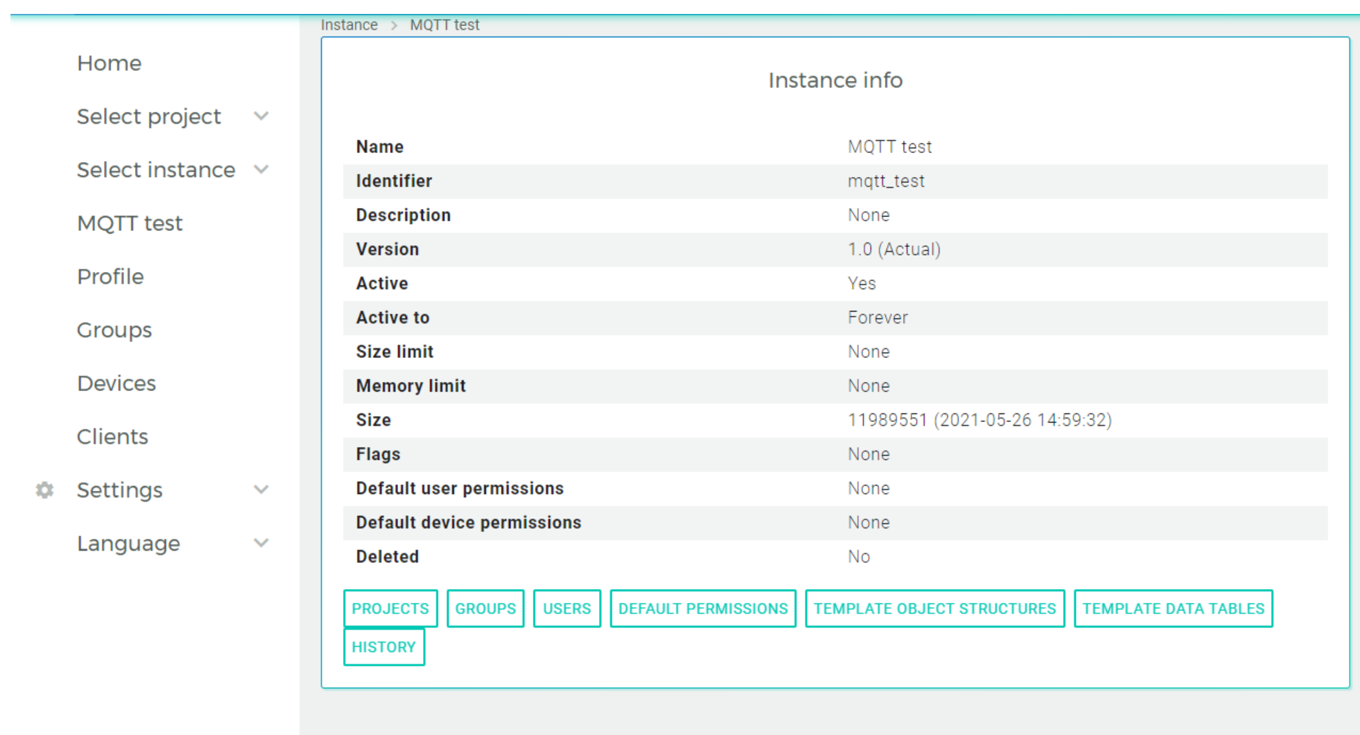


Obrázek 5.1: Host ve webovém rozhraní Cloudu

- Name: jméno hostu.
- Identifier: identifikátor hostu.
- OS: operační systém.

- Internal hostname: IP adresa, v našem případě localhost. Adresa, na kterou se připojujeme přes konfigurátor.
- External hostname: IP adresa v případě snahy o připojení cizí aplikace zvenku (obejít připravený worker).
- Connect to database with SSL: pokud existuje více hostů, je lepší používat šifrování. U localhostu není nutné.
- Database port: port databáze.
- Catalog database: jméno catalogové databáze, pokud existuje (není nutná).
- Default tablespace path: možnost změny místa, kam se ukládají data.
- Size limit: omezení paměti na disku.
- Memory limit: omezení paměti procesoru, v našem případě None (bez omezení).
- Active: ukazuje, zda je host aktivní.
- Connection status: stav připojení.

Poté je třeba vygenerovat instanci na daném hostu. To může dělat pouze administrátor. Při vytvoření instance dojde automaticky k vytvoření databáze v PostgreSQL (timescaleDB), viz kapitola 2.7. Vygenerovaná instance pro tento experiment je zobrazena na Obrázku 5.2.



Obrázek 5.2: Instance ve webovém rozhraní Cloudu

- Name: jméno instance.
- Identifier: identifikátor instance.
- Description: stručný popis instance.

- Version: verze instance.
- Active: ukazuje, zda je instance aktivní.
- Active to: možnost nastavení, do kdy bude instance aktivní, v našem případě není omezení, proto hodnota forever.
- Size limit: omezení paměti na disku.
- Memory limit: omezení paměti procesoru, v našem případě None (bez omezení).
- Size: současná velikost.
- Flags: flagy sloužící k dalšímu nastavení instance.
- Default user permissions: přístupová práva která dostane jakýkoliv uživatel automaticky.
- Default device permissions: přístupová práva která dostane jakékoliv zařízení automaticky.
- Deleted: ukazuje, zda je projekt smazán.

Dále je třeba vygenerovat projekt na této instanci. Jedna instance může obsahovat více projektů. Všechny projekty sdílí stejnou databázi. Námi vytvořený projekt je znázorněn na Obrázku 5.3.

Project > test

Home
Select project
test
Object structu...
Objects
Select instance
MQTT test
Profile
Groups
Devices
Clients
Settings
Language

Project info

Name	test
Identifier	test
Description	test
Version	1.0 (Actual)
Active	Yes
Active to	Forever
Size limit	None
Memory limit	None
Collation	en-US-x-icu
Size	376832 (2021-05-26 14:59:32)
Flags	None
Default user permissions	None
Default device permissions	None
Deleted	No

OBJECTS OBJECT STRUCTURES
ROLES TRANSLATION LANGUAGES DEFAULT PERMISSIONS TEMPLATE OBJECT STRUCTURES TEMPLATE DATA TABLES
HISTORY EDIT DELETE

Obrázek 5.3: Projekt ve webovém rozhraní Cloudu

Zde jsou zobrazeny informace o projektu, zároveň je zde možnost editace vlastností tohoto projektu.

- Name: jméno projektu.
- Identifier: identifikátor projektu.

- Description: stručný popis projektu.
- Version: verze projektu.
- Active: ukazuje, zda je projekt aktivní.
- Active to: možnost nastavení, do kdy bude projekt aktivní, v našem případě není omezení, proto hodnota forever.
- Size limit: omezení paměti na disku.
- Memory limit: omezení paměti procesoru, v našem případě None (bez omezení).
- Collation: nastavení, jak řadit sloupečky v projektu (české řazení, anglické řazení).
- Flags: flagy sloužící k dalšímu nastavení projektu.
- Default user permissions: přístupová práva která dostane jakýkoliv uživatel automaticky.
- Default device permissions: přístupová práva která dostane jakékoliv zařízení automaticky.
- Deleted: ukazuje, zda je projekt smazán.

V tomto projektu je třeba vytvořit objektovou tabulku a do té přidat sloupeček Name. Sloupečky data_table_id, id a deleted jsou vygenerovány automaticky. Naše testovací objektová tabulka je zobrazena na Obrázku 5.4. A její sloupečky na Obrázku 5.5.

Obrázek 5.4: Objektová struktura ve webovém rozhraní Cloudu

- ID: ID objektové tabulky.
- Display name: jméno tabulky.
- Description: popis tabulky.

- Parent object structure: odkaz na rodičovskou strukturu.
- Flags: flagy sloužící k dalšímu nastavení tabulky.
- Template version: ukazuje verzi template, pomocí kterého byla tabulka vytvořena (v našem případě nebyl template použit, proto hodnota none).
- Generated: ukazuje, zda je tabulka vygenerována nebo ne.
- Visible: udává, zda je tabulka viditelná či nikoliv.
- Locked: udává, zda je tabulka zamčená.
- Archived: ukazuje, zda je tabulka archivována.
- Memory limit: omezení paměti procesoru, v našem případě None (bez omezení).
- Size limit: omezení paměti na disku.
- Size: současná velikost.
- Deleted: Ukazuje, zda je tabulka smazána nebo ne.

Object structure columns list

Find FILTER CLEAR

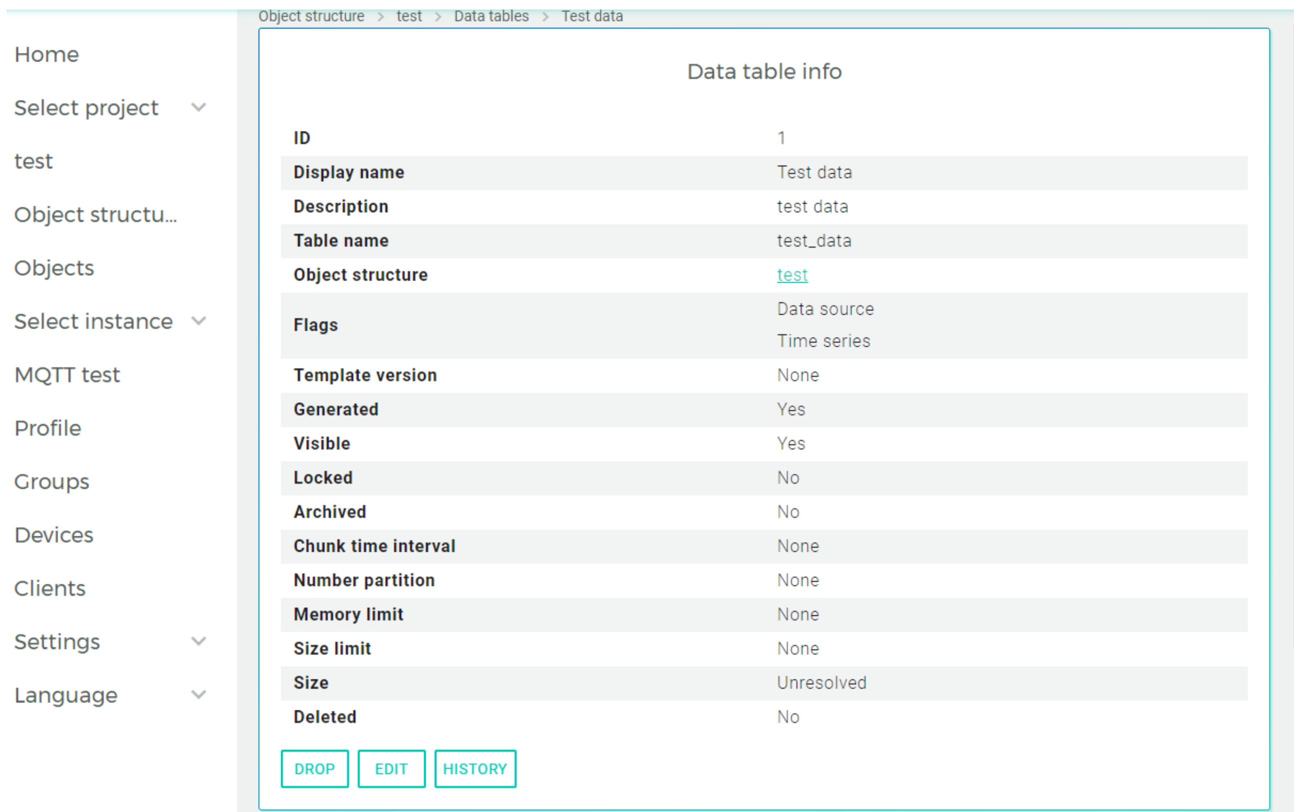
Column name	Display name	Type	Data type	Flags	Unique combination	Detail
id	ID	Identifier	Integer	Autoincrement	None	
data_table_id	Data Table ID	Data Table Identifier	Integer	None	None	
name	Name	Display name	String	None	None	DETAIL
deleted	Deleted	Deleted	Boolean	None	None	

SHOW DELETED

Obrázek 5.5: Sloupečky objektové tabulky ve webovém rozhraní Cloudu

- id: ID objektu.
- data table id: ID datové tabulky, do které se budou ukládat data objektu.
- name: jméno.
- deleted: Ukazuje, zda je tabulka smazána nebo ne.

Dále je potřeba v této objektové struktuře vytvořit datovou tabulku. Na tuto tabulku mohou být navázány jednotlivé objekty přes jejich data_table_id. Do této tabulky se pak ukládají reálná data. Příklad vygenerované tabulky z experimentu je uveden na Obrázku 5.6.



Obrázek 5.6: Datová tabulka ve webovém rozhraní Cloudu

Při vytváření datové tabulky je možné upravit tyto její následující parametry:

- ID: ID datové tabulky.
- Display name: jméno tabulky.
- Description: popis tabulky.
- Table name: interní jméno tabulky.
- Object structure: struktura, do které daná tabulka patří.
- Flags: flagy sloužící k dalšímu nastavení tabulky.
- Template version: ukazuje verzi template, pomocí kterého byla tabulka vytvořena (v našem případě nebyl template použit, proto hodnota none).
- Generated: ukazuje, zda je tabulka vygenerována nebo ne.
- Visible: udává, zda je tabulka viditelná či nikoliv.
- Locked: udává, zda je tabulka zamčená.
- Archived: ukazuje, zda je tabulka archivována.
- Memory limit: omezení paměti procesoru, v našem případě None (bez omezení).
- Size limit: omezení paměti na disku.
- Size: současná velikost.

- Deleted: Ukazuje, zda je tabulka smazána nebo ne.

Dále je také možné zobrazit si historii této tabulky. Pro fungování workeru je třeba do tabulky přidat sloupečky time a value, které budou představovat čas a uloženou hodnotu.

Column name	Display name	Type	Data type	Flags	Detail
object_id	Object ID	Object Identifier	Integer	None	
individual_id	Individual ID	Author Identifier	Integer	None	
time	Time	Primary time	Datetime	None	DETAIL
value	Value	Major analog value	Double	None	DETAIL

SHOW DELETED

Obrázek 5.7: Sloupečky datové tabulky ve webovém rozhraní Cloudu

- object id: ID daného objektu.
- individual id: ID uživatele nebo zařízení, které hodnotu zapsalo.
- Time: čas, který přišel spolu s hodnotou.
- Value: uložená hodnota.

Následně je nutné vytvořit konkrétní objekt vytvořené objektové tabulky a nastavit mu vygenerovanou datovou tabulku jako data table. ID objektové tabulky a ID konkrétního objektu jsou pak použity v rámci poslané MQTT zprávy.

Object structure > test > Objects > Object 1

Object info

ID	1
Table ID	1
Name	Object 1
Data table	Test data
Deleted	No

CHILDS ROLES HISTORY EDIT DATA TABLE EDIT DELETE

Obrázek 5.8: Objekt v datové tabulce ve webovém rozhraní Cloudu

- ID: ID daného objektu.
- table ID: ID objektové tabulky, které náleží tento objekt.
- Name: jméno objektu.
- Data table: jméno datové tabulky.
- Deleted: Ukazuje zda je tabulka smazána nebo ne.

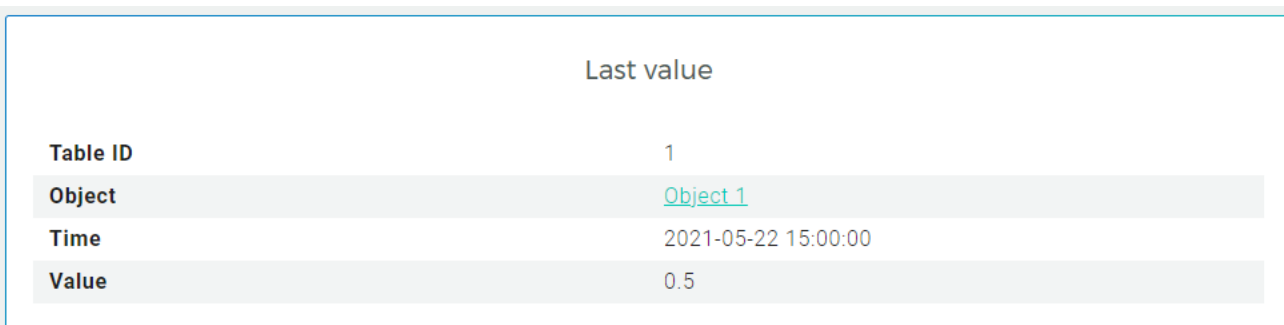
Pro ověření funkčnosti byl celý program spuštěn na serveru a pomocí jednoduchého publisheru byla odeslána testovací data. Zároveň bylo otestováno i založení více služeb naráz.

Tetovací data poslaná pomocí MQTT publisheru byla strukturována v JSON formátu následovně:

```
msg = {
      "apiKey": "abcd",
      "secretKey": "abcd",
      "project": "test",
      "objectTableId": "1",
      "objectId": "1",
      "data":
      {
         "time": "2021-05-22 15:00:00",
         "value": "0.5"
      }
}
```

Za apiKey a secretKey je třeba dosadit konkrétní klíče vygenerované pomocí Cloudu. Project reprezentuje jméno projektu, ve kterém se nacházejí dané tabulky, objectTableId (ID objektové tabulky) a objectID (ID daného objektu) slouží k identifikaci objektu, do jehož datové tabulky chceme zapsat přiložená data. Time je čas naměřené hodnoty a value je naměřená hodnota.

Následně bylo ve webovém rozhraní ověřeno, že požadovaná data byla ve správném formátu uložena do správné tabulky.



Obrázek 5.9: Zobrazení posledního přidaného záznamu ve webovém rozhraní

- Table ID: ID objektové tabulky.
- Object: jméno objektu.
- Time: čas, který je součástí přijaté zprávy.
- Value: uložená hodnota, která byla přijata spolu s časem Time.

Výsledky ukazují, že data jsou správně uložena do předem vytvořené tabulky. Pro lepší pochopení si můžeme představit teoretickou situaci. Chceme například monitorovat stroje v několika výrobních halách. Každá hala může mít různé části a každá část několik strojů. Postup bude následující:

- Nejprve se zaregistruje host a na něm se vytvoří instance (databáze), všechny stroje budou mít data uloženy v této databázi (je i možnost pro každou halu vytvořit vlastní instanci).
- Dále se vytvoří projekt (pokud existuje více komplexů, může například jeden projekt reprezentovat jeden komplex).
- Vytvoří se objektová tabulka, ve které budou jednotlivé haly. Na tuto tabulku se může navázat další objektová tabulka, kde budou části haly. Na tu lze navázat tabulku strojů.
- Na tabulku strojů už lze navázat datovou tabulku, do které se budou ukládat data.

Každý stroj má tedy svojí datovou tabulku. Zároveň je každý stroj potomkem objektu představujícího část haly, ten je zase potomkem objektu představujícího konkrétní halu a ten teprve náleží nějakému projektu. Toto je samozřejmě jen jeden z mnoha způsobů. Klient si může hierarchii udělat podle svých potřeb. Například nemusí existovat žádná hierarchie a všechny stroje mohou být uloženy pouze v jedné objektové tabulce na stejné úrovni. Toto nastavení už závisí na preferencích daného zákazníka a je zcela libovolné.

Tímto způsobem lze uchovávat jednotlivé informace o strojích (teplota, tlak, spotřeba energie, doba provozu) k pozdějšímu použití a prozkoumání.

5.2 Budoucí vývoj

V tomto momentě aplikace funguje, ale je zde pár věcí, které by bylo dobré doplnit a vylepšit.

- Je třeba zajistit bezpečnost proti takzvanému SQL injection útoku. Jde o útok, který pomocí neošetřeného vstupu dokáže nahradit select svým vlastním kódem, což může vést k úniku citlivých dat.
- Dále je na několika místech cachovat (uchovávat některá data v paměti pro pozdější rychlejší přístup) connection, místo toho aby bylo pokaždé voláno znova.
- Dalším vylepšením by bylo zpracování binárních dat. Pro to by bylo nutné zjišťovat datový typ sloupečku a v případě binárních dat je konvertovat z textu do binárního formátu pomocí base64.
- Bylo by dobré provést testování na nějakém reálném systému a ujistit se, že vše funguje, jak má.
- Je třeba se ještě zamyslet nad autorizací. Ta byla z větší části ignorována, protože byla příliš složitá a pro první fázi programu nepotřebná.

Kapitola 6

Závěr

Tato práce se zabývala návrhem a vývojem workeru, který slouží k zapisování dat do databáze. Během této práce byly popsány použité technologie MQTT, IoT, Edge computing, Cloud computing a jeho služby, Cloudové úložiště, postgreSQL, REST a JSON-RPC. Dále byl představen IoT Cloud, s kterým bude worker spolupracovat. Byla navržena architektura daného workeru, která využívá MQTT protokol a slouží jako prostředník mezi klientem a IoT databází.

Po seznámení s protokolem MQTT a možnostmi Cloudu byla navržena architektura, která splňovala všechny požadavky. Dále proběhlo testování nastavení Cloudu a brokeru. Poté byly pomocí navržené architektury vytvořeny dvě třídy v jazyce Python, které představují samotný worker. Nejprve bylo testováno vytváření nových subprocesů a vláken. Poté následovala část, kdy bylo potřeba vymyslet, jak bude broker reagovat na případné změny instancí (ať už změny portu nebo vytvoření, popřípadě odstranění nějaké instance). Tato část zabrala spoustu času, protože bylo nutné docílit toho, aby se všechny vlákna a subprocesy řádně ukončily. Následně proběhl test celého workeru na serveru. Tento test ukázal nějaké další chyby, které bylo třeba opravit. Nakonec byl proveden finální test, kdy bylo otestováno připojení k databázi a uložení zaslaných dat. Zároveň byla testována komunikace s dvěma publishery najednou. Tento test proběhl úspěšně. I přesto, že je stále pár věcí, které lze vylepšit, aby byl worker plně použitelný, worker splňuje svůj hlavní účel.

Literatura

[1] Mohamed Abdel-Basset, Mai Mohamed, and Victor Chang. Nmcda: A framework for evaluating cloud computing services. *Future Generation Computer Systems*, 86:12–29, 2018.

[2] Corinne Bernstein, Kate Brush, and Alexander S. Gillis. What is mqtt and how does it work?, Jan 2021. URL: `https://internetofthingsagenda.techtarget.com/definition/MQTT-MQ-Telemetry-Transport`.

[3] What is a cloud service? – cloud service definition - citrix. URL: `https://www.citrix.com/glossary/what-is-a-cloud-service.html`.

[4] Korry Douglas and Susan Douglas. *PostgreSQL: a comprehensive guide to building, programming, and administering PostgresSQL databases*. SAMS publishing, 2003.

[5] Eric Hamilton. What is edge computing: The network edge explained, 12 2018. URL: `https://www.cloudwards.net/what-is-edge-computing/`.

[6] Minhaj Ahmad Khan and Khaled Salah. Iot security: Review, blockchain solutions, and open challenges. *Future Generation Computer Systems*, 82:395–411, 2018.

[7] Rob Kiefer. Timescaledb vs. postgresql for time-series, Nov 2020. URL: `https://blog.timescale.com/blog/timescaledb-vs-6a696248104e/`.

[8] Martin Malý. Protokol mqtt: komunikační standard pro iot, Jun 2016. URL: `https://www.root.cz/clanky/protokol-mqtt-komunikacni-standard-pro-iot/`.

[9] What is rest. URL: `https://restfulapi.net/`.

[10] REX Controls s.r.o. *MQTT driver for the REXYGEN system (the MQTTDrv module) – User guide*. Pilsen, Czech Republic, 2.50.10 edition, 12 2020. Accessed: 2021-05-24. URL: `https://www.rexygen.com/doc/ENGLISH/MANUALS/MQTTDrv/MQTTDrv_ENG.html`.

[11] Jiyi Wu, Lingdi Ping, Xiaoping Ge, Ya Wang, and Jianqing Fu. Cloud storage as the infrastructure of cloud computing. In *2010 International Conference on Intelligent Computing and Cognitive Informatics*, pages 380–383, 2010. `doi:10.1109/ICICCI.2010.119`.

[12] Rpc 2.0 specification. URL: `https://www.jsonrpc.org/specification`.

[13] Postgresql. URL: `https://www.postgresql.org/about/`.

[14] Zhi-Kai Zhang, Michael Cheng Yi Cho, and Shiuhpyng Shieh. Emerging security threats and countermeasures in iot. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, page 1–6, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2714576.2737091`.

Seznam použitých zkratek

MQTT: Message Queuing Telemetry Transport.
IoT: Internet of things.
JSON-RPC: JavaScript Object Notation - remote procedure call.
SaaS: Software as a Service.
IaaS: Infrastructure as a Service.
PaaS: Platform as a Service.
StaaS: Storage as a Service.
DB: Database.
HTTP: Hypertext Transfer Protokol.
M2M: Machine to machine.
ISO: International Organization for Standardization.
IEC: International Electrotechnical Commission.
IP: Internet Protocol.
SQL: Structured Query Language.
RFC: Request for Comments.
CPU: central processing unit.
SCADA: Supervisory control and data acquisition.
TCP/IP: Transmission Control Protocol/Internet Protocol .
API: Application programming interface.
OS: Operating system.
SSH: Secure Shell.
OPC UA: Open Platform Communications United Architecture.
SSL/TLS: Secure Sockets Layer/Transport Layer Security.