# ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

**Fakulta elektrotechnická**

**Katedra počítačů**

## Univerzální enterprise IoT síť

## Universal enterprise IoT network

Diplomová práce

Studijní program: Otevřená Informatika

Studijní obor: Softwarové inženýrství

Vedoucí bakalářské práce: Ing. Peter Macejko

**Bc. Jan Zubr**

Praha 2021

## PODĚKOVÁNÍ

Děkuji Ing. Peterovi Macejkovi za vedení diplomové práce, odborný pohled, cenné podněty a připomínky, které v rámci vedení poskytoval. Dále děkuji i rodině za její podporu v průběhu psaní práce.

# PROHLÁŠENÍ

Prohlašuji, že jsem bakalářskou práci s názvem „*Univerzální enterprise IoT sít*“ vypracoval samostatně a použil k tomu odbornou literaturu a prameny, které uvádím v seznamu přiloženém k bakalářské práci.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V *Praze* dne ……………… ……………………….

Podpis

**Název diplomové práce:** Univerzální enterprise IoT síť


## Abstrakt:

Tato práce má za cíl navrhnout IoT síť zaměřenou na použití v enterprise prostředí, která je modulární, efektivní, bezpečná, rozšiřitelná a lze ji integrovat s ostatními systémy. Práce navazuje na předcházející bakalářkou práci. V úvodu jsou popsány nedostatky předcházející práce a na jejich základě definovány nové cíle. Je provedena analýza aktuálních možností pro řešení IoT systémů a na jejím základě je proveden návrh IoT systému, resp. sítě. Následně je popsána implementace Proof-of-Concept – PoC aplikace v jazyce Java, která je otestována a porovnána s předcházejícím řešením.



## Klíčová slova:

distribuované systémy, Internet věcí, redundance, cluster, middleware, Java


**Master´s Thesis title:** Universal enterprise IoT network


## Abstract:

This work aims to design an IoT network for use in an enterprise environment that is modular, efficient, secure, scalable and can be integrated with other systems. The work builds on the previous bachelor thesis. The introduction describes the shortcomings of the previous work and on the basis of them new goals are defined. An analysis of current possibilities for solving IoT systems is performed followed by a design of enterprise IoT system. Subsequently, the implementation of Proof-of-Concept - PoC application in Java is described, which is tested and compared with the previous solution.



## Key words:

distributed systems, Internet of Things, redundancy, cluster, middleware, Java

# Obsah

**1 Úvod ... 12**
1.1 Motivace ... 12
1.2 Cíle práce ... 13
1.3 Problémy předcházející verze ... 13
1.3.1 Administrativní náročnost ... 14
1.3.2 Rozšiřitelnost logiky ... 14
1.3.3 Formát zpráv závislý na jazyce ... 15
1.3.4 Mohutnost minimálního nasazení ... 15
1.4 Nové požadavky na systém ... 15
1.4.1 Nasazení lokálně a v cloudu ... 15
1.4.2 Změna logiky za běhu ... 15
1.4.3 Jednotné schéma zpráv ... 16
1.4.4 Centralizovaný i decentralizovaný provoz ... 16
**2 Možnosti realizace IoT systémů ... 17**
2.1 Logické modely pro IoT systémy ... 17
2.1.1 Client – Server ... 17
2.1.2 Master – Slave ... 18
2.1.3 Publish – Subscribe ... 18
2.1.4 Peer – to – Peer ... 18
2.2 Modely dle umístění business logiky ... 19
2.2.1 V místě – On premises ... 19
2.2.2 Hybdridní – Edge computing ... 19
2.2.3 Vzdáleně – Cloud ... 19
2.2.3.1 Kolokace ... 20
2.2.3.2 Hosting ... 20
2.2.3.3 Infrastructure as a Service – IaaS ... 20
2.2.3.4 Platform as a Service – PaaS ... 20
2.2.3.5 Software as a Service – SaaS ... 20
2.3 Middleware pro IoT systémy ... 21
2.3.1 RabbitMQ ... 21

2.3.2 ActiveMQ ..... 21
2.3.3 KubeMQ ..... 22
2.3.4 Apache Kafka ..... 22
2.3.5 Apache Pulsar ..... 22
2.4 Komunikace a její formát v rámci IoT systémů ..... 23
2.4.1 Apache Avro ..... 23
2.4.2 Apache Thrift ..... 24
2.4.3 Protocol Buffers ..... 24
2.5 Existující platformy pro IoT systémy ..... 24
2.5.1 Node-RED ..... 24
2.5.2 Thingsboard.io ..... 25
2.5.3 AWS IoT ..... 25
2.5.4 Azure IoT ..... 26
2.5.5 Google IoT ..... 26
2.5.6 Cisco IoT ..... 27
2.5.7 IBM Watson IoT Platform ..... 27
2.6 Blockchain pro IoT ..... 27
2.6.1 IOTA ..... 28
2.6.2 Helium ..... 29
2.6.3 IoTeX ..... 30
**3 Návrh řešení ..... 31**
3.1 Funkční požadavky ..... 31
3.1.1 Návrh systému jako IoT platformy ..... 31
3.1.2 Široká možnost integrace s jinými systémy ..... 31
3.1.3 Využití mikro-servisní architektury ..... 31
3.1.4 Redundance a vysoká dostupnost kritických součástí ..... 31
3.1.5 Nasazení v cloudu, on premise nebo obojí zároveň ..... 31
3.1.6 Funkce v koordinovaném a nekoordinovaném režimu ..... 31
3.1.7 Existence speciálního koordinačního uzlu ..... 32
3.1.8 Provoz v koordinovaném a nekoordinovaném režimu současně ..... 32
3.1.9 Možnost přepínání uzlů mezi režimy ..... 32
3.1.10 Perzistentní unikátní ID uzlů ..... 32
3.1.11 Neperzistentní unikátní adresy uzlů ..... 32

3.1.12 Mapování mezi adresou a ID uzlu ... 32
3.2 Architektura ... 32
3.2.1 Volba technologií ... 33
3.2.2 Obecná architektura platformy ... 34
3.2.3 Obecný model uzlu ... 35
3.2.4 Adresace a pojmenování uzlů ... 36
3.2.5 Model komunikace ... 37
3.2.6 Koordinovaný a nekoordinovaný mód ... 38
**4 Implementace ... 40**
4.1 Komunikační protokol ... 40
4.1.1 Přenos hodnot ... 41
4.1.2 Komunikace mezi uzly ... 42
4.1.2.1 Zpráva s popisem uzlu – NodeDescriptor ... 42
4.1.2.2 Zpráva s konfigurací uzlu – NodeConfig ... 43
4.1.3 Vzájemná detekce uzlů a koordinátorů ... 43
4.1.4 Autorizace koordinátora ... 44
4.2 Převod Avro IDL na Java balík ... 44
4.3 Jádro uzlu ... 45
4.3.1 Služba pro správu proměnných ... 46
4.3.2 Služba komunikace ... 46
4.3.3 Řízení stavu ... 47
4.3.4 Procesní služby ... 47
4.3.5 Implementace proměnných ... 47
4.4 Implementace uzlu ... 48
4.4.1 Stavový automat uzlu ... 48
4.4.1.1 Stav připojování – Joining ... 48
4.4.1.2 Nekoordinovaný stav – Non-Coordinated ... 49
4.4.1.3 Koordinovaný stav – Coordinated ... 49
4.4.1.4 Odmítnutý stav – Rejected ... 49
4.4.2 Uzel UART (RS-232, RS-422, RS-485) ... 50
4.4.3 Uzel řídící logiky – kontrolér radiátoru ... 50
4.4.4 Uzel InfluxDB Sink ... 51
4.5 Implementace koordinátora ... 52

4.5.1 Služba pro detekci uzlů – NodeService ... 53
4.5.2 Služba pro správu konfigurací – NodeConfigStorage ... 53
4.5.3 Služba pro koordinované uzly – CoordinatedNodeService ... 53
4.6 Zabezpečení ... 54
4.6.1 Zabezpečení Apache Pulsar ... 54
4.6.1.1 Zabezpečení uzlů ... 55
4.6.1.2 Zabezpečení koordinátorů ... 55
**5 Nasazení a testování ... 56**
5.1 Testovací prostředí ... 56
5.1.1 Server ... 56
5.1.2 Síťová infrastruktura ... 57
5.2 Nasazení Apache Pulsar ... 57
5.3 Nasazení uzlů ... 58
5.3.1 Uzel UART (RS-232, RS-422, RS-485) ... 58
5.3.2 Uzel řídící logiky – kontrolér ... 59
5.3.3 Uzel InfluxDB Sink ... 59
5.4 Testování ... 59
5.4.1 Využití systémových prostředků ... 59
5.4.2 Objem komunikace ... 60
5.4.2.1 Výpočet velikosti objektu Value ... 60
5.4.2.2 Příklad a porovnání ... 61
5.4.2.3 Naměřená data ... 63
5.4.3 Stabilita ... 64
5.4.4 Bezpečnost ... 64
5.4.4.1 Test přístupu k Apache Pulsar – Autentifikace ... 64
5.4.4.2 Test autorizace uzlu ... 65
5.4.4.3 Test autorizace koordinátora ... 65
**6 Závěr ... 66**
**Seznam použité literatury ... 68**
**Seznam zkratek ... 70**
**Příloha A: Seznam elektronických příloh ... 71**

# Seznam obrázků

Obrázek 1 Architektura logického uzlu ..... 14
Obrázek 2 IOTA Tangle ..... 29
Obrázek 3 IoTeX architektura ..... 30
Obrázek 4 Příklad Flow-Based Programming ..... 34
Obrázek 5 Příklad realizace v uvedené architektuře ..... 35
Obrázek 6 Obecný model uzlu ..... 36
Obrázek 7 Příklad komunikace – výměna hodnot proměnných ..... 38
Obrázek 8 Hlídání stavu koordinátora uzlem ..... 39
Obrázek 9 Závislost částí projektu ..... 40
Obrázek 10 Schéma převodu AVDL na jar balík ..... 45
Obrázek 11 Přibližná podoba jádra ..... 46
Obrázek 12 Stavový automat uzlu ..... 50
Obrázek 13 Schéma nasazení ..... 56
Obrázek 14 Logické schéma PoC aplikace ..... 58
Obrázek 15 Stavová zpráva rozhraní – příklad ..... 62
Obrázek 16 Serializovaný objekt Value ..... 62
Obrázek 17 Architektura systému z bakalářské práce ..... 65

# 1 Úvod

*Počátky Internetu lze datovat už do 60. let 20. století, ale forma, ve které jej známe dnes, začala vznikat spojováním separovaných sítí a utvářením standardů v 80. letech téhož století. Internet přinesl do oblasti počítačů a informační techniky řadu nových možností a výzev, se kterými bylo potřeba se vypořádat. Tyto skutečnosti umožnily lidem navzájem komunikovat a vytvářet systémy, které jsou přes internet provázány a taktéž spolu komunikují.*

*V posledních letech zaznamenáváme významný nárůst tzv. chytrých zařízení, která se k Internetu připojují a vyměňují si jeho prostřednictvím data. Příkladem mohou být garážová vrata ovládaná přes internetový prohlížeč. Celkovým trendem vývoje je vytvořit z těchto chytrých zařízení provázaný ekosystém, označovaný jako „Internet věcí“, též známý pod anglickým pojmem Internet-of-Things (IoT).*

*V tomto ekosystému si zařízení připojená k Internetu navzájem vyměňují data a podle nich reagují, přizpůsobují se, učí se a vzájemně se integrují. Jako příklad lze uvést dům, který uzavře okna a uzamkne dveře, pokud se jeho obyvatelé nenacházejí v blízkosti. Zde se logicky musí jednat o souhru několika akcí, které jsou vykonány různými členy systému na základě nějaké události.*

*Způsob výměny informací včetně bezpečnosti přenosu mezi jednotlivými členy obecného IoT systému představuje novou výzvu, se kterou je potřeba se vypořádat. Existuje velmi mnoho možností, jakými lze k řešení této výzvy přistoupit. Vzhledem k rozmanitosti zařízení, která mohou v IoT sítích komunikovat, nelze zvolit jeden, ten „nejlepší“ způsob komunikace. Je třeba brát ohledy na výkon zařízení, energetickou spotřebu, konektivitu a celkové finanční náklady na jeho provoz.* [1]

V rámci této diplomové práce navazuji na svou bakalářskou práci z roku 2019. Zaměřím se zejména na zhodnocení výsledků praktického nasazení předcházejícího systému, popis problémů a jejich řešení s ohledem na aktuální trendy a dostupné technologie. Výstupem pak bude architektonický návrh a základní implementace druhé generace universální IoT sítě.

## 1.1 Motivace

Během praktického nasazení systému navrženého v bakalářské práci se objevilo několik zásadních problémů, které budou blíže rozebrány v kapitole 2. Ty se v testovacích podmínkách

nejevily příliš kritické nebo se neprojevil jejich negativní dopad na funkčnost. Po jejich bližším prozkoumání se ovšem ukázalo, že příčina těchto problémů leží v samotném návrhu. Navíc se ukázaly nové funkční požadavky, jejichž implementace je v rámci předcházející práce velmi obtížná nebo dokonce nemožná.

Z výše uvedených příčin vyplývá, že je nutné provést zásadní změny v architektuře celého systému. Ta by měla podporovat snadnou integraci s ostatními systémy a zároveň umožňovat do budoucna snadnou rozšiřitelnost o nové, dosud neexistující funkcionality.

## 1.2 Cíle práce

V rámci této práce bude provedena rešerše a analýza existujících řešení pro obecné IoT systémy a technologií, které jsou pro IoT použitelné. Na základě těchto výsledků budou vybrány vhodné technologie pro navrhovaný systém. V širším kontextu jsou cíle této práce velmi podobné jako v předcházející bakalářské práci.

*Cílem této práce je navrhnout univerzální komunikační model pro lokální IoT síť. Důraz bude kladen hlavně na efektivitu přenosu dat v rámci sítě tak, aby užitečná data byla co nejmenší a zabírala co největší podíl v rámci přenášeného bloku. Dále se v návrhu modelu zaměřím na zabezpečení této IoT sítě, aby nehrozil únik dat či manipulace se síti. Důležitým cílem návrhu je eliminace jediného bodu selhání (Single-Point-Of-Failure, SPOF), aby síť zůstala v provozu i při vyřazení několika klíčových prvků. Toto položí základ pro vznik distribuovaného systému využívajícího principy vysoké dostupnosti se značnou mírou rozšiřitelnosti a přizpůsobitelnosti.* [1]

Výše uvedené cíle jsou stále platné i pro tuto práci a jsou dále rozšířeny. Budou vyřešeny problémy a zakomponovány nové požadavky detailně rozebírané v kapitole 2. Navržený systém bude implementován, otestován a porovnán s původním systémem v oblasti modularity a objemu přenesených dat. Výsledky testování nového systému budou analyzovány s ohledem na stanovené cíle.

## 1.3 Problémy předcházející verze

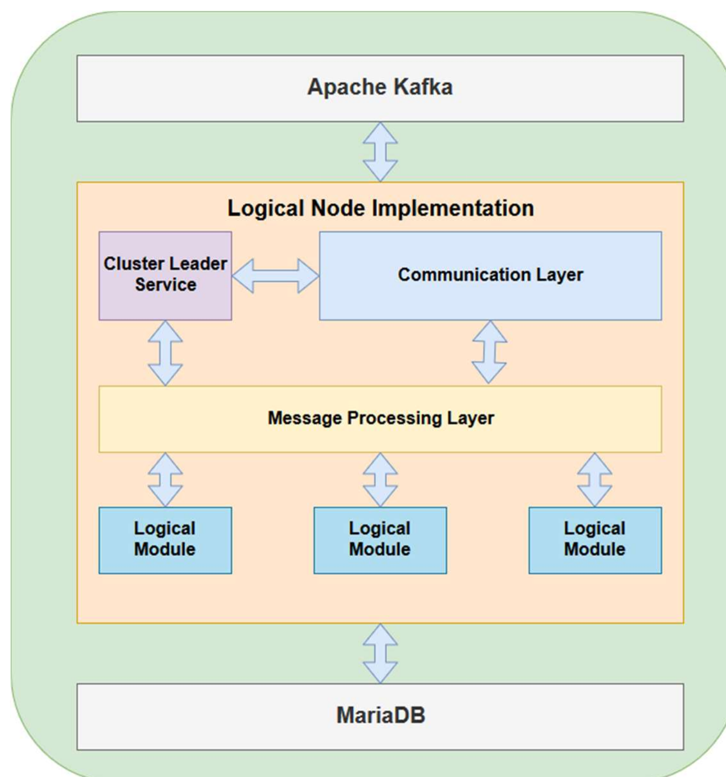
Předchozí verze zpracovaná v bakalářské práci má následující zásadní nedostatky. Vysoké nároky na administrativní správu celého systému, velmi obtížnou rozšiřitelnost o novou logiku, přílišnou závislost formátu přenášených zpráv na programovacím jazyce Java a mohutnost minimálního nasazení.

### 1.3.1 Administrativní náročnost

Problém administrativní náročnosti spočívá ve škálovatelnosti systému. Původní systém je sám o sobě velmi dobře horizontálně škálovatelný v oblasti Kafka brokerů a logických uzlů. Nicméně tato škálovatelnost je problematická v případě uzlů Zookeeper a databázových uzlů Galera clusteru. Zde je vyžadováno, aby při přidání dalšího uzlu byla IP adresa nebo hostname, nově přidávaného uzlu vloženy do konfiguračních souborů všech ostatních uzlů. Toto pak může v některých případech vyžadovat restart celého Zookeeper nebo Galera clusteru, aby upravená konfigurace byla načtena.

### 1.3.2 Rozšiřitelnost logiky

Obtížná rozšiřitelnost logiky celého systému se ukázala jako zásadní nedostatek, neboť tato logika byla součástí aplikace logického uzlu. Ten měl monolitickou podobu, zobrazenou na obrázku 1. Při horizontálním škálování to vyžadovalo, aby všechny logické uzly byly vždy kompilovány ze stejné verze zdrojového kódu. Tento problém se v praktickém nasazení experimentálně potlačoval postupným restartem (tzv. rolling-restart), ale docházelo k nepředvídatelnému chování celého systému během nekonzistentního stavu, výjimečně i k poškození či ztrátě dat. Alternativou bylo všechny logické uzly restartovat zároveň, ale tím byla narušena podmínka vysoké dostupnosti.




**Obrázek 1 Architektura logického uzlu (Zdroj: Zubr, Jan, Generická IoT síť, ČVUT FEL Praha 2019)**

### 1.3.3 Formát zpráv závislý na jazyce

V původním systému je formát zpráv definován pomocí Java tříd a neexistuje přenositelné schéma, které by umožnilo snadnou implementaci v jiném programovacím jazyce. To znamená, že při přidání modulu zpracovávajícího výše uvedené zprávy by bylo nutné získat schéma manuálně z Java třídy. Takový postup je nevhodný pro multiplatformní systém implementovaný v několika různých jazycích.

### 1.3.4 Mohutnost minimálního nasazení

Dalším zásadním problém zjištěným během praktického nasazení původního systému je celková mohutnost služeb, které musejí být zajištěny pro minimální možnou funkci. Bylo nutno nasadit instance Zookeeper, Apache Kafka brokeru, MariaDB a logického uzlu. To je velmi nepraktické, pokud jde o propojení dvou fyzických uzlů mezi sebou bez nutnosti přidané logiky nebo ukládání dat. Postačovalo by jen nasadit Zookeeper a Kafka broker. Což však původní systém neumožňuje.

## 1.4 Nové požadavky na systém

Ze zmíněných problémů vyplývají nové požadavky na celý systém. Detailně budou rozpracovány v kapitole 3.1 Funkční požadavky. Zde uvedu jen přehled těch klíčových, které dávají návrhu nového systému pevný rámec.

Nasazení jednotlivých částí systému jak lokálně, tak i v cloudu, dále možnost rozšiřovat a měnit logiku za běhu, na jazyku nezávislé schéma přenášených zpráv a možnost funkce v centralizovaném i decentralizovaném režimu zároveň.

### 1.4.1 Nasazení lokálně a v cloudu

Systém musí podporovat nasazení všech částí pomocí kontejnerizační technologie, jako je například Docker[1], OpenShift[2] nebo Kubernetes[3]. Toto umožní nasadit části systému jak lokálně i v cloudu, případně oba přístupy volně kombinovat.

### 1.4.2 Změna logiky za běhu

V rámci systému musí být možno změnit jednotlivé části logiky za běhu bez dopadu na jeho ostatní části. Logika tedy musí být modulární.

---

[1] https://www.docker.com/
[2] https://www.redhat.com/en/technologies/cloud-computing/openshift
[3] https://kubernetes.io/

### 1.4.3 Jednotné schéma zpráv

Zprávy vyměňované v systému budou mít své schéma definované bez závislosti na programovacím jazyku tak, aby implementace jakékoliv části systému v libovolném jazyce mohla zprávy podle tohoto schématu serializovat a deserializovat.

### 1.4.4 Centralizovaný i decentralizovaný provoz

Systém musí být schopen funkce v režimu centrálního řízení a zároveň v režimu bez něho. To znamená, že některé jeho části budou koordinované centrálně a další budou zcela autonomní. V systému také mohou existovat části, které podporují přechod mezi režimy řízení za běhu.

# 2 Možnosti realizace IoT systémů

IoT systémy jsou ve své podstatě speciálním případem distribuovaných systémů, proto u nich můžeme pozorovat společné charakteristiky. Těmi jsou zapojení několika uzlů, vzájemná komunikace mezi nimi a možnost využití vysoké dostupnosti.

Při návrhu a realizaci IoT systémů je tedy logické vycházet z principů využívaných u distribuovaných systémů. Jedná se zejména o volbu modelu architektury. Je nutno nalézt vhodný architektonický model k zamýšlenému užití celého systému. To samo o sobě představuje značnou výzvu, protože cílem této práce je, aby systém byl co nejvíce univerzální.

Existuje široká paleta architektonických modelů, které lze rozdělit dle způsobu komunikace mezi uzly a dle umístění business logiky. Tato dvě dělení jsou na sobě navzájem nezávislá a lze je mezi sebou kombinovat.

Dále je nutno při návrhu IoT systému zvážit způsob výměny dat mezi jeho uzly. Na výměnu dat existuje široké množství protokolů a formátů. Při volbě protokolu je důležité zohlednit jeho otevřenost a zároveň i výpočetní možnosti všech uzlů systému. Některé protokoly či formáty vhodné například pro komunikaci mezi aplikacemi na serverech nejsou vhodné pro komunikaci mezi IoT uzly. Ty nemusejí mít dostatek výpočetního výkonu ke zpracování zprávy poslané přes tento protokol.

## 2.1 Logické modely pro IoT systémy

Následující modely byly již uvedeny v předcházející práci. Považuji však za vhodné zopakovat celý jejich přehled a doplnit drobné korekce. Neuvedení přehledu by výrazně snížilo kvalitu práce, protože text se na něj bude často odkazovat.

### 2.1.1 Client – Server

V modelu Client – Server zahajuje vždy komunikaci klient vůči serveru. Server zpracovává požadavky klienta a odesílá mu zpět odpověď. Ve většině případů jde o aplikaci obsahující business logiku či poskytující nějakou službu klientům. Velká výhoda tohoto modelu spočívá v tom, že je použitelný pro aplikace nasazené v prostředí, kde nemáme kontrolu nad síťovou infrastrukturou. Zde není překážkou překlad IP adres (NAT). Jedinou nutnou podmínkou je, dostupnost severu pro všechny klienty, například pomocí veřejné IP adresy. Tento komunikační model je široce využíván u IoT systémů dodávaných jako služba, například Google Home, bezpečnostní systém SimplySafe., běžná databáze či webová aplikace. [2]

### 2.1.2 Master – Slave

Model Master – Slave je ve své podstatě logickým opakem modelu Client – Server. Uzel, který je v roli master, zahajuje, řídí a ukončuje veškerou komunikaci se svými podřízenými uzly, které jsou v roli slave. Tohoto modelu využívá například databázové úložiště, kdy jedna instance databáze je označena jako master a ostatní jako slave. Master instance do slave instancí replikuje změny. [3]

### 2.1.3 Publish – Subscribe

V tomto modelu existuje prostředník, případně skupina prostředníků – brokeři, se kterými uzly komunikují. Uzly se u brokerů registrují do komunikačních kanálů – témat a určují, zda dané téma chtějí odebírat, či do něj publikovat. Publikující uzel odesílá brokerovi zprávu s identifikací tématu. Ten tuto zprávu distribuuje všem uzlům, které dané téma odebírají, a zároveň i dalším brokerům, pokud je jich v systému více. Výhoda tohoto modelu spočívá v možnosti jednotlivých uzlů libovolně mezi sebou navazovat komunikaci a provozovat ji synchronně i asynchronně. Komunikace je vždy vedena přes brokery, takže jednotlivé uzly mezi sebou nemají přímé spojení. Tento model sdílí množství charakteristik s modelem Client – Server a je tedy také vhodný pro nasazení v prostředí s minimální či žádnou kontrolou nad síťovou infrastrukturou. Jediná podmínka je, aby broker byl dostupný pro všechny uzly stejně, jako je server pro klienty. Nicméně broker, který vykonává roli serveru, je univerzální a jedna instance může být použita i pro více různých aplikací. Tento model využívají různé middleware aplikace, například Apache Kafka. [2]

### 2.1.4 Peer – to – Peer

Model Peer – to – Peer je model, u kterého nelze určit jeden centralizovaný prvek, jak to bylo možné u všech předcházejících modelů. Zde může přímo komunikovat každý uzel s každým, podobně jako v Publish – Subscribe. Chybí však centralizovaný broker a jednotlivé uzly v tomto modelu fungují jako klient a server zároveň. Velkou výhodou tohoto modelu je odolnost proti výpadkům jednotlivých uzlů, ale implementace takového systému je náročná. Další zásadní nevýhodou je problematická funkce v prostředí, kde se vyskytuje NAT. Tento model vyžaduje, aby každý uzel byl dostupný pro všechny ostatní. Na principu peer – to – peer je založena známá síť pro sdílení souborů BitTorrent. [2]

## 2.2 Modely dle umístění business logiky

Modely můžeme také rozdělit dle umístění business logiky. Existují tři možnosti, kde logiku nasadit. V místě, kde je systém provozován, na vzdáleném serveru – cloud, nebo lze využít hybridního přístupu označovaného jako Edge computing.

### 2.2.1 V místě – On premises

Umístění celé logiky systému v místě provozu je tradiční pojetí nasazení. Provozovatel musí zajistit vše nezbytné pro bezchybný provoz. Velmi často to znamená, že musí mít k dispozici serverovnu, IT administrační personál pro správu serverovny, musí zajistit nákup hardware a software, instalace aplikací a operačních systémů. Jedná se o nejnáročnější model nasazení s ohledem na finanční, prostorové, bezpečnostní a personální požadavky. Tyto požadavky lze částečně omezit pomocí outsourcingu IT personálu a vybavení.

### 2.2.2 Hybdridní – Edge computing

Hybridní nasazení, též známé jako Edge computing, je model, kdy se jedna část logiky nachází na lokálním serveru (Edge server nebo Edge node) a druhá část logiky na vzdáleném. Tento hybridní model poskytuje několik zásadních výhod oproti nasazení pouze na vzdáleném serveru. Jeho pomocí lze minimalizovat objem přenesených dat po internetové lince, odezvu systému v mission-critcal případech a eliminovat závislost na naprosté spolehlivosti internetové linky. V případě výpadku konektivity může systém v omezeném režimu fungovat dál. [4]

Personální, finanční, bezpečnostní a prostorové požadavky mohou být na stejné úrovni jako u nasazení v místě. Tuto zátěž provozovatele lze omezit. Jednou z možností je outsourcing IT oddělení. Poskytovatel systému nebo služby může také k provozovateli (zákazníkovi) umístit vlastní hardware a zajistit jeho správu. To je někdy označováno jako Hardware as a Service (HaaS).

### 2.2.3 Vzdáleně – Cloud

Vzdálené nasazení spočívá v umístění business logiky mimo místo provozu systému. Většinou zde dochází k využívání internetových služeb třetích stran. Existuje celkem 5 úrovní vzdáleného nasazení. První dvě jsou kolokace a hosting. Jsou to o úrovně, které předcházely vzniku cloudu, proto je vhodné se o nich zmínit. Další tři úrovně jsou již souhrnně označovány jako cloud. Jedná se o poskytování infrastruktury, platformy nebo software jako služby.

Nejznámějšími poskytovateli cloudových služeb jsou Microsoft Azure[4], Amazon AWS[5] nebo IBM Cloud[6].

#### 2.2.3.1 Kolokace

Kolokace, též známá jako server housing nebo serverhosting, znamená, že jako služba je zákazníkovi poskytována část datacentra spolu s internetovou konektivitou a elektrickou energií. Zákazník si musí zajistit nákup hardware, jeho instalaci a správu uvnitř datacentra.

#### 2.2.3.2 Hosting

Při využití hostingu si zákazník pronajímá fyzický nebo virtuální server, který je majetkem poskytovatele. Správu serveru si musí zákazník zajistit sám, nebo mu může být nabídnuta v různém rozsahu jako služba ze strany poskytovatele. Výhodou je v takovém případě pravidelná bezplatná obměna hardware.

#### 2.2.3.3 Infrastructure as a Service – IaaS

V rámci IaaS je jako služba poskytována infrastruktura v podobě virtuálních serverů, úložiště, sítě a virtualizace. Poskytovatel služby je vlastníkem infrastruktury a se stará o její chod a údržbu. Odběratel má nad poskytnutou infrastrukturou v podobě virtuální strojů plnou kontrolu a je sám zodpovědný za instalaci operačního systému, včetně všech požadovaných aplikací. Výhodou IaaS je velká škálovatelnost, protože virtuální servery lze snadno upravovat díky tomu, že dochází k pronájmu hardware zdrojů, jako jsou jádra procesoru, RAM a úložiště. [5]

#### 2.2.3.4 Platform as a Service – PaaS

PaaS představuje další úroveň cloudových služeb a staví na IaaS. Zde poskytovatel v rámci služby dodává operační systém a další podpůrný software. Zákazník musí pouze nainstalovat jeho aplikace a naplnit je daty. Výhoda PaaS spočívá v tom, že vývojář pracující s tímto modelem je osvobozen od správy operačního systému a služeb, což umožňuje jednodušší a rychlejší vývoj aplikací. [5]

#### 2.2.3.5 Software as a Service – SaaS

Model SaaS je pravděpodobně nejčastější model cloudové služby, se kterou se můžeme v běžném životě setkat. Poskytovatel služby dodává kompletní řešení v podobě softwaru

[4] https://azure.microsoft.com
[5] https://aws.amazon.com
[6] https://www.ibm.com/cloud

dostupného přes internet. Zákazník se stará pouze o data, kterými svou instanci naplní. Výhodou SaaS je komplexní řešení, eliminace potřeby vývoje a údržby aplikace ze strany zákazníka, který platí pouze paušální poplatek dle velikosti jeho instance. [5]

## 2.3 Middleware pro IoT systémy

Komplexní distribuovaný nebo IoT systém implicitně vyžaduje, aby jednotlivé uzly mohly spolu komunikovat a vyměňovat si zprávy. Toto řeší specializovaný integrační software, souhrnně označovaný middleware, který si lze představit jako trubky propojující jednotlivé aplikace. Volba komunikačního middleware je při návrhu IoT systému zcela zásadní. Musí být škálovatelný, poskytovat nízkou odezvu, vysokou dostupnost a podporu pro široké spektrum komunikačních formátů.

Následující výčet obsahuje nejznámější zástupce komunikačního middleware. Ti splňují základní požadavky vysoce spolehlivého IoT systému, jako je vysoká dostupnost, podpora Publish – Subsribe komunikace, nezávislost na formátu přenášených dat a možnost provozu v cloudu či lokálně pomocí kontejnerů, jako je Docker, OpenShift nebo Kubernetes. Proto budou u jednotlivých zástupců rozebrány pouze klíčové vlastnosti, které je odlišují od ostatních.

### 2.3.1 RabbitMQ

RabbitMQ byl původně navržen pro použití protokolu AMQP[7]. Později byla přidána podpora pro protokoly STOMP[8] a MQTT[9]. AMQP je tedy hlavní protokol. RabbitMQ dále umožňuje výměnu zpráv přes HTTP a WebSocket. [6]

Další vlastností RabbitMQ je podpora směrování zpráv na základě směrovacího klíče. Díky němu může producent zprávu doručit pouze k určitým konzumentům. Tento systém směrování využívá zástupné znaky a lze tak vytvořit hierarchickou strukturu klíčů. Konzument tak může odebírat velmi malou přesně definovanou podmnožinu zpráv. [7]

### 2.3.2 ActiveMQ

ActiveMQ využívá jako svůj základ Java Message Service – JMS. Díky tomu je možné ActiveMQ broker integrovat přímo do Java aplikace, a vyhnout se tak nutnosti nasazení samostatného brokera. Jedná se open source projekt. ActiveMQ podporuje nasazení v clusteru, čímž dosahuje škálovatelnosti a vysoké dostupnosti. Umožňuje směrování zpráv pomocí

[7] ISO/IEC 19464:2014. Information technology -- Advanced Message Queuing Protocol (AMQP) v1.0
[8] https://stomp.github.io/
[9] ISO/IEC 20922:2016. Information technology -- Message Queuing Telemetry Transport (MQTT) v3.1.1

zástupných symbolů, čímž lze vytvořit hierarchickou strukturu stejně jako u RabbitMQ. Podporovanými protokoly jsou OpenWire[10], STOMP, AMQP a MQTT. [8]

### 2.3.3 KubeMQ

KubeMQ se orientuje primárně na nasazení v Kubernetes a OpenShift, ale lze jej provozovat i v Dockeru. Díky tomuto modelu může být cluster nasazen tak, že některé uzly jsou on premise a některé v cloudu, případně ve více cloudech. Stejně jako RabbitMQ a ActiveMQ podporuje hierarchickou strukturu a lze tak zprávy směrovat mezi uzly. [9]

Specifikem KubeMQ je fakt, že se jedná o platformu, která je poskytována jako služba. Pro nasazení KubeMQ brokera je potřeba registrace a registrační token. V rámci licence poskytované zdarma je možný provoz pouze jedné instance. [10]

### 2.3.4 Apache Kafka

Apache Kafka se od předchozích zástupců zásadně liší. Jedná se o streamovací platformu a interně využívá distribuovaný log. Kafka pracuje s textovými soubory uloženými na souborovém systému brokera a zprávy zapisuje na jejich konec. Díky velice jednoduché architektuře má Kafka vysokou propustnost a nízkou odezvu. V těchto parametrech je jednoznačně lepší než RabbitMQ a ActiveMQ [11]. Negativní dopad těchto výhod však je, že nepodporuje směrování zpráv. Konzumenti tak musejí filtraci zpráv v rámci tématu provádět sami. Pro svou funkci Kafka vyžaduje napojení na Apache Zookeeper cluster, který je využíván pro volbu vůdce, správu konfigurací témat, řízení přístupu a udržování seznamu aktivních brokerů. To znamená nasazení další služby, kterou je potřeba udržovat. Nasazení Apache Kafka tedy není tak jednoduché, jako v případě RabbitMQ, ActiveMQ nebo KubeMQ. [12]

### 2.3.5 Apache Pulsar

Apache Pulsar je z uvedených middleware řešení nejkomplexnější. Je to distribuovaná platforma pro zasílání zpráv a streamování zaměřená na cloudové prostředí. Nejzajímavějšími vlastnostmi této platformy je bezstavový charakter jednotlivých brokerů a nativní podpora bezserverového vykonání kódu.

Bezstavový charakter brokeru umožňuje rapidně škálovat celý systém horizontálně bez zásadního dopadu na výkon. Takto lze cluster operativně škálovat bez nutnosti synchronizace dat při přidání nebo odebrání brokera.

[10] https://activemq.apache.org/openwire

Nativní podpora bezserverového vykonání kódu je zásadní odlišností od ostatních zástupců. To znamená, že lze využít Pulsar cluster pro vykonávání business logiky. Ta nemusí být nasazena na vlastním serveru nebo v kontejneru, a má tak stejnou úroveň redundance jako samotný cluster. Pulsar podporuje bezserverové vykonání kódu v jazycích Java, Python a Go. Dále podporuje integraci s jazyky C++, C# a JavaScript. Lze tedy očekávat v blízké době podporu bezserverového kódu i pro ně. [13]

## 2.4 Komunikace a její formát v rámci IoT systémů

Pro komunikaci mezi systémy je potřeba přenášet i velmi komplexní datové struktury. Aby se daly přenést, využívá se principu serializace. Datová struktura je převedena na posloupnost bajtů, které jsou následně odeslány příjemci.

Při volbě formátu komunikace i serializace v rámci IoT systému je potřeba zohlednit, jaká zařízení se budou na komunikaci podílet, jak je rozšířený a podporovaný daný formát a kolik úsilí stojí jej implementovat a později udržovat. Existuje mnoho možností serializace, jako například textové protokoly YAML Ain't Markup Language – YAML, JavaScript Object Notation – JSON a Extensible Markup Language – XML či binární Protocol Buffers (zkráceně Protobuf) a Apache Avro.

Binární protokoly nabízejí oproti textovým výraznou úsporu přenesených dat. Jsou však člověkem nečitelné, a tak je orientace v nich velice obtížná, při komplexnějších zprávách nemožná. Čitelnost dat pro člověka ale není pro IoT systém zásadní, protože se jedná hlavně o Machine-to-Machine – M2M komunikaci. Proto budou rozebrány pouze binární komunikační protokoly.

### 2.4.1 Apache Avro

Apache Avro je kompletní serializační systém. Poskytuje možnost vytvářet složité datové struktury, rychlou binární serializaci a deserializaci, možnost ukládání do souborů, vzdálené volání procedur – RPC a generování kódu.

Avro využívá konceptu schémat. Ty jsou běžně ve formátu JSON a fungují jako návod pro serializaci a deserializaci dat. Protože samotné schéma ve formátu JSON může být velmi komplexní, poskytuje Avro vlastní jednoduchý jazyk Avro Interface Description Language – Avro IDL, který je kompilován dodávaným nástrojem do JSON schémat, nebo do zdrojového kódu jazyku Java, C++ a C#.

Tento protokol je velmi podobný svou funkcionalitou Protocol Buffers a Thirft. Zásadní odlišností Apache Avro je dynamické typování. V dynamicky typovaných jazycích není nutné generovat kód a lze rovnou využít JSON schéma. Díky tomu je možno v těchto jazycích měnit schéma za běhu programu. U přenášených dat nedochází k označování datovým typem a dalšími informacemi. Jsou přenášeny pouze hodnoty, což znamená značnou úsporu. [14]

### 2.4.2 Apache Thrift

Apache Thrift stejně jako Apache Avro je kompletní serializační systém, který podporuje generování kódu a vzdálené volání procedur. Oba systémy jsou si velmi podobné svými funkcemi. Apache Thrift je oproti Apache Avro staticky typovaný. To znamená, že musí být vygenerován kód a schéma nelze měnit za běhu. Výhodou Apache Thrift je širší paleta podporovaných jazyků. [15]

### 2.4.3 Protocol Buffers

Protocol Buffers byl navržen v roce 2001 Googlem a v roce 2008 uvolněn jako open source. Jedná se o velmi robustní systém, který byl původně vyvíjen pro jazyk C++. Z toho důvodu využívá statické typování stejně jako Apache Thrift.

Disponuje vlastním jazykem pro definici formátu zpráv a podporuje generování kódu do jazyků C++, Java, Dart, Go, Python, Ruby, C#, JavaScript a PHP. Za výhodu Protocol Buffers oproti Avru lze považovat tagovaná data. Díky tomu je zjednodušená aktualizace komunikačního protokolu napříč systémy, protože tagy umožňují detekci změn v protokolu za běhu aplikace. Nedochází tak k chybám při deserializaci, a tím je celý systém výrazně robustnější. [16]

## 2.5 Existující platformy pro IoT systémy

IoT má v současnosti velkou popularitu a postupně se dostává do našeho každodenního života. Proto existují řešení, která se mu věnují. Některá jsou dostupná jako open source a na jejich vývoji se podílí aktivní komunita, jiná jsou čistě komerční. Nejviditelnějšími poskytovateli komerčních komplexních IoT řešení jsou Amazon, Google a Microsoft. Dalšími, méně známými, jsou Cisco a IBM.

### 2.5.1 Node-RED

Node-RED je open source nástroj pro propojování aplikačních rozhraní – API, online služeb a hardwarových zařízení. Je vytvořený v JavaScriptu a běží pod Node.js. Disponuje grafickým editorem, ve kterém se logika programuje pomocí toku dat. Toto programovací

paradigma se běžně označuje jako Flow-based programming – FBP. Node-RED je uživatelsky velmi přívětivý, populární a je využíván i v průmyslovém nasazení. Velkou výhodou je rozšiřitelnost, takže není problém přidávat nové funkce z komunitních repositářů. [17]

Toto řešení však nepodporuje vysokou dostupnost, takže jej nelze provozovat v clusteru. Existuje návrh[11], který sice umožňuje provoz více instancí Node-RED zároveň, ale nejedná se o příliš standartní řešení. Náročnost na správu a chybovost může být příliš vysoká.

### 2.5.2 Thingsboard.io

Thingsboard je ucelená platforma pro realizaci IoT řešení. Je dostupná v open source bezplatné komunitní a zpoplatněné profesionální verzi. Obě verze se od sebe liší v tom, jaké funkce poskytují. V komunitní verzi není k dispozici například plánovač, integrace na standartní protokoly a možnost exportovat reporty a data do souborů.

Stejně jako Node-RED i Thingsboard využívá FBP přístup. Na rozdíl od Node-RED je možno Thingsboard nasadit jako cluster, a to lokálně nebo v cloudu. Dále podporuje rozšiřitelnost některých částí pomocí skriptů napsaných v jazyce Python. [18]

### 2.5.3 AWS IoT

Společnost Amazon v rámci své cloudové platformy Amazon Web Services – AWS nabízí i řešení pro IoT systémy – AWS IoT. IoT zařízení mohou pro komunikaci využít HTTP, MQTT nebo WebSockets. AWS IoT poskytuje možnost napojení na další AWS cloud služby, autentizaci zařízení a vývojové prostředí.

Dále je součástí platformy také AWS IoT GreenGrass[12], což je software, který je určen pro edge computing. Díky tomu lze některé funkce IoT systému přenést z cloudu do lokálního zařízení a uchovat tak částečnou funkcionalitu systému i při nespolehlivé konektivitě.

Zajímavostí je Amazonem vyvíjený FreeRTOS[13]. Jedná se o real-time operační systém pro embedded zařízení a lze jej využít i na levných vývojových deskách založených například na čipech ESP32 nebo STM32. Díky tomu lze AWS IoT integrovat snadno do široké škály projektů.

[11] https://discourse.nodered.org/t/how-do-i-achieve-clustering-and-load-balancing-with-node-red-native-capabilities/16672/7
[12] https://aws.amazon.com/greengrass/
[13] https://aws.amazon.com/freertos

Amazon k této platformě poskytuje roční bezplatný trial. Zpoplatnění celé platformy je nastaveno velmi atraktivně. Cena roste v inkrementech po jednom milionu operací a pohybuje se od 0.08USD do 1.25USD za milion provedených operací[14]. [19]

### 2.5.4 Azure IoT

Platforma Azure IoT od společnosti Microsoft je poskytovaná v rámci komplexního cloudového řešení Azure. Umožňuje komunikaci pomocí HTTP, MQTT, AMQP a WebSockets. Podobně jako AWS IoT poskytuje autentizaci zařízení, jejich management a monitoring.

Azure IoT také nabízí možnost přenesení části cloudové funkcionality do lokálního zařízení. To zajišťuje software IoT Edge. Je poskytován zdarma a je nedílnou součástí celého Azure IoT.

Dostupná zcela zdarma je také značná část celé cloudové platformy Azure, včetně IoT služeb. Zde je však omezení na 8000 zpráv/zařízení/den s velikostí do 500 bajtů. Při překročení této hranice se poté platí za každé zařízení od 25 USD za měsíc[15]. [20]

### 2.5.5 Google IoT

Google je další významný poskytovatel cloudových služeb, jejichž součástí je i řešení pro IoT. Pro komunikaci s IoT zařízeními lze využít HTTP nebo MQTT. Tato platforma se skládá ze dvou hlavních částí. Device Manager umožňuje správu, autentizaci a řízení připojených IoT zařízení. Protocol Bridge se stará o zpracování zpráv z MQTT a HTTP a jejich následnou publikaci do Cloud Pub/Sub služby. Z ní mohou být poté zprávy směrovány do dalších služeb Google Cloud.

Na rozdíl od Amazonu nebo Microsoftu nenabízí Google možnost edge computingu v podobě dodávaného softwarového řešení. Pro využití edge computing je nutno zakoupit zařízení obsahující Google Edge TPU[16] chip.

Cenová politika Google Cloud IoT je založena na přenesených datech. Prvních 250 MB dat je zdarma a poté se platí od 0.00045 USD do 0.0045 USD za každý přenesený MB[17]. [21]

[14] https://aws.amazon.com/iot-core/pricing/?loc=ft#AWS_Free_Tier_with_AWS_IoT_Core
[15] https://azure.microsoft.com/cs-cz/pricing/details/iot-hub/
[16] https://cloud.google.com/edge-tpu
[17] https://cloud.google.com/iot/pricing

### 2.5.6 Cisco IoT

Společnost Cisco je také poskytovatelem vlastní IoT platformy. Je zaměřena zejména na oblast mobilních sítí. Jediná podporovaná komunikace je pomocí MQTT. V rámci své platformy poskytuje Cisco jako službu eSIM, 5G konektivitu a strojové učení.

Je také k dispozici edge computing. Cisco spolupracuje s Microsoftem na integraci s Azure, čímž by mělo dojít k možnosti toku dat mezi oběma platformami[18].

Cenová politika zde není známa a není poskytována jakákoliv bezplatná verze služeb. [22]

### 2.5.7 IBM Watson IoT Platform

IBM Watson IoT Platform je další alternativou k řešením od společností Google, Amazon a Microsoft. Také poskytuje služby správy zařízení a možnost využívání dalších cloudových služeb v rámci IBM Cloud. Specifikem IBM platformy je zaměření na kombinaci umělé inteligence, IoT a BigData do jednoho celku označovaného jako Digitální dvojče, Digital Twin. Dále IBM Watson IoT Platform nabízí nativní využití technologie blockchain, což je v rámci ostatních IoT platforem neobvyklé. Komunikace zařízení s platformou je možná opět pomocí MQTT nebo HTTP. [23]

IBM v rámci Watson IoT nabízí zdarma Lite plán, který je limitován na 500 zařízení, 500 aplikačních spojení a 200 MB dat přenesených za měsíc[19]. Další plány nejsou známy, jsou pravděpodobně nabízeny formou cenové nabídky na základě poptávky.

## 2.6 Blockchain pro IoT

Blockchain je systém zaznamenávání informací takovým způsobem, který činí neoprávněnou změnu nebo podvod v systému obtížnými nebo zcela nemožnými. Jedná se v podstatě o digitální účetní knihu transakcí, která je duplikována a distribuována po celé síti počítačových uzlů v blockchainu. Každý blok v řetězci obsahuje určitý počet transakcí. Pokaždé, když v blockchainu dojde k nové transakci, je záznam o ní přidán do hlavní knihy každého účastníka. Decentralizovaná databáze spravovaná více účastníky se nazývá Distributed Ledger – DL.

[18] https://blogs.cisco.com/internet-of-things/cisco-and-microsoft-partner-to-enable-seamless-data-orchestration-from-cisco-iot-edge-to-azure-iot-cloud
[19] https://cloud.ibm.com/catalog/services/internet-of-things-platform

Technologie založené na blockchainu, resp. DL v posledních letech získávají na popularitě a jsou k dispozici i řešení pro IoT. Blockchain se v IoT využívá pro M2M komunikaci a ukládání dat, a může tak sloužit jako decentralizovaná alternativa ke cloudovým službám uvedeným v kapitole 2.5. Mohou existovat případy, kdy je upřednostnění této technologie před centralizovaným řešením ekonomicky výhodnější a poskytuje vyšší přidanou hodnotu.

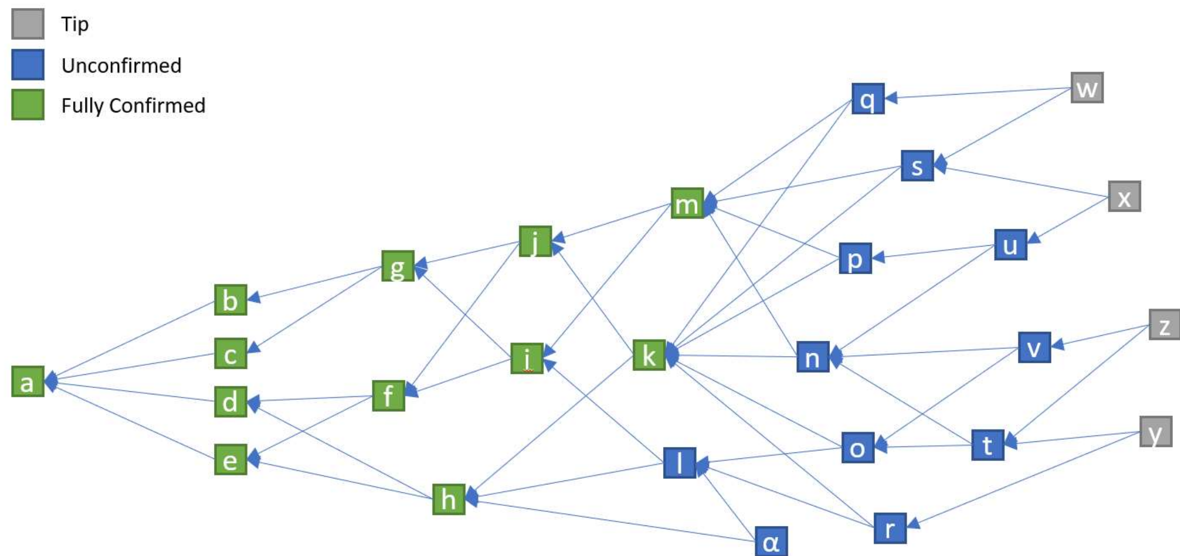
Navzdory obrovskému potenciálu IoT je stále jeho adopce zpomalována překážkami, jako je nedostatečná škálovatelnost, obavy o soukromí, důvěryhodnost a ukládání dat nebo centralizaci těchto dat u jediné společnosti. Dále IoT v současné době postrádá smysluplný způsob realizace přidané hodnoty. Tedy skutečné komunikace mezi jednotlivými zařízeními spolu s využitím business logiky, a to plně decentralizovaně. Centralizované modely provozu většinou znamenají „být pouze připojen“ k jednomu bodu, kde se veškerá přidaná hodnota nachází.

Řešení těchto problémů může představovat blockchain navržený specificky pro IoT. Ten lze využít pro vytvoření zabezpečeného privátního IoT řešení, nad již existujícím základem. Blockchainové sítě typicky obsahují velké množství uzlů a tvoří decentralizovaný systém, který disponuje vysokou dostupností. Data jsou chráněny kryptograficky a nakládat s nimi může pouze ten, kdo disponuje příslušnými privátními klíči. Business logiku lze do blockchainové sítě nasadit a mít tak zajištěno, že se vykoná vždy.

### 2.6.1 IOTA

IOTA (MIOTA) je DL určený k zaznamenávání a provádění transakcí mezi stroji a zařízeními v Internetu věcí. Transakce se v síti provádějí pomocí kryptoměny s názvem MIOTA ($10^6$ IOTA). Klíčovou inovací IOTA je Tangle. Nejedná se o klasický blockchain ve smyslu řetězce bloků. Tangle je orientovaný acyklický graf, kde každá transakce odkazuje na dvě předcházející. Toto řešení umožňuje vysokou škálovatelnost a zároveň paralelní zpracovávání transakcí. Každý uzel, který do Tangle přidává novou transakci musí dvě předcházející zvalidovat. Tento princip je zobrazen na obrázku 2. Tím, že je uzel sítě zároveň validátor, je eliminována nutnost těžby měny a poplatků v síti. Jednotlivé uzly sítě tak mohou komunikovat a ukládat data zdarma. [24]

Síť IOTA disponuje širokou paletou knihoven. Podporovány jsou jazyky Java, Rust, Python a JavaScript[20]. Komunikace probíhá přes HTTPS protokol. Potvrzení transakcí v IOTA síti není náročné a existují i implementace pro mikroprocesory[21].




**Obrázek 2 IOTA Tangle (Zdroj https://github.com/noneymous/iota-consensus-presentation/blob/master/README.md)**

## 2.6.2 Helium

Helium je komplexní řešení pro IoT, které je open-source a open-hardware. Helium síť se skládá ze 3 částí: Helium Hotspot, Helium Console a Helium Blockchain.

Helium Hotspoty jsou podobné WiFi access point – AP zařízením. Místo WiFi signálu ale vysílají „LongFi“ signál založený na LoRaWAN[22] bezdrátovém protokolu. Toto umožňuje Hotspotu pokrýt oblast o poloměru několika kilometrů, na rozdíl od WiFi, která dokáže pokrýt pouze desítky metrů. Hotspoty slouží jako brány mezi běžným internetem a LoRaWAN Helium sítí.

Helium Console umožňuje přístup k Helium síti, managment a získání dat z připojených zařízení a integraci na další služby, jako je například Microsoft Azure.

Helium Blockchain slouží pro zápis a čtení dat z připojených zařízení. To je zpoplatněno použitím kryptoměny Helium, která je dána jako odměna hotspotům a validátorům za zajištění provozu sítě a pokrytí oblasti LongFi signálem. [25]

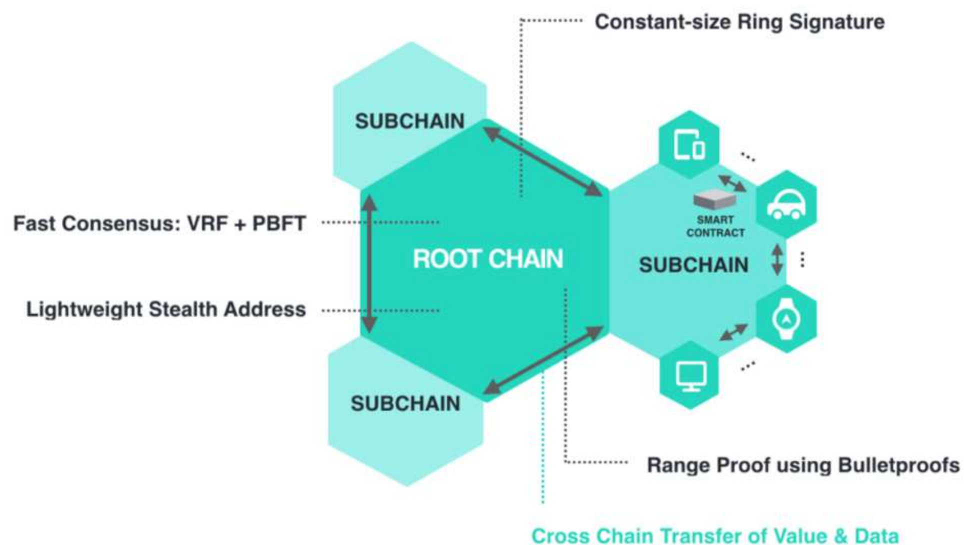
[20] https://wiki.iota.org/iota.rs/getting_started
[21] https://blog.iota.org/iota-esp32-wallet-1b12b45d8a5/
[22] https://lora-alliance.org/about-lorawan/

Aktuálně je po celém světě v provozu přes 400 tisíc aktivních hotspotů, zejména ve velkých městech. Helium sítí je pokryta celá Praha. Toho využívá například společnost Lime pro monitoring svých sdílených koloběžek[23].

### 2.6.3 IoTeX

IoTeX (IOTX) je řešení „blockchain-within-a-blockchain“ pro IoT a jeho cílem je fungovat jako distribuované komunikační médium. Je navržen tak, aby umožnil propojení IoT zařízení, byl rychlý a škálovatelný. IoTeX také zajišťuje ochranu soukromí v rámci přenášených zpráv a realizovaných transakcí.

Architektura IoTeX se skládá z kořenového blockchainu, který poskytuje zabezpečení a správu sítě a zároveň spravuje další sub-blockchainy. Tuto architekturu zobrazuje obrázek 3. Dílčí blockchainy se používají k připojení IoT zařízení, která fungují v podobných prostředích, mají podobnou úroveň důvěryhodnosti nebo podobný funkční účel. Zároveň je umožněna komunikace mezi jednotlivými sub-blockchainy. To má několik výhod. Jednou z nich je, že kořenový blockchain zůstane nedotčen při napadení jakéhokoliv sub-blockchainu. Další výhodou je škálovatelnost. [26]




**Obrázek 3 IoTeX architektura (Zdroj https://www.liskmagazine.com/blog/2019/07/18/elitex-launches-staking-rewards-for-iotex-where-blockchain-meets-iot/)**

[23] https://www.helium.com/ecosystem

# 3 Návrh řešení

## 3.1 Funkční požadavky

Při návrhu každého systému je nutné definovat funkční požadavky. Ty jsou popisem služeb a funkcionalit, které musí software poskytovat a popisují formou funkce buď celý systém, nebo jeho jednotlivé komponenty. Funkcí je myšlena kombinace vstupu, chování a výstupu. Může jít o výpočet, manipulaci s daty, business proces, uživatelskou interakci nebo jakoukoli jinou specifickou funkcionalitu, kterou může systém vykonávat. Funkční požadavky se někdy také označují jako Funkční specifikace.

Pro tento systém je na základě zkušeností z předchozí bakalářské práce stanoven následující seznam funkčních požadavků.

### 3.1.1 Návrh systému jako IoT platformy

Systém bude navržen jako IoT platforma umožňující flexibilní výstavbu IoT řešení.

### 3.1.2 Široká možnost integrace s jinými systémy

Systém bude umožňovat snadnou a širokou integraci na ostatní existující systémy.

### 3.1.3 Využití mikro-servisní architektury

Systém bude využívat mikro-servisní architektury.

### 3.1.4 Redundance a vysoká dostupnost kritických součástí

Systém bude redundantní a vysoce dostupný v kritických částech tak, aby nehrozil kompletní výpadek.

### 3.1.5 Nasazení v cloudu, on premise nebo obojí zároveň

Systém bude možno nasadit v cloudu, lokálně či kombinovaně bez náročných změn v konfiguraci jednotlivých částí.

### 3.1.6 Funkce v koordinovaném a nekoordinovaném režimu

Systém bude umožňovat provoz uzlů v koordinovaném režimu s hlavním koordinačním uzlem i v nekoordinovaném režimu bez koordinačního uzlu.

### 3.1.7 Existence speciálního koordinačního uzlu

V sítí může existovat speciální typ uzlu – koordinátor, který bude obstarávat koordinaci běžných uzlů. V síti může být více koordinátorů zároveň. Každý z nich bude spravovat svou skupinu uzlů.

### 3.1.8 Provoz v koordinovaném a nekoordinovaném režimu současně

Systém bude umožňovat souběžný provoz uzlů v koordinovaném i nekoordinovaném režimu.

### 3.1.9 Možnost přepínání uzlů mezi režimy

Uzel systému může být za běhu přepnut z nekoordinovaného režimu do koordinovaného režimu a naopak.

### 3.1.10 Perzistentní unikátní ID uzlů

Každý uzel bude mít vlastní unikátní perzistentní ID.

### 3.1.11 Neperzistentní unikátní adresy uzlů

Každý uzel bude při svém startu generovat neperzistentní unikátní adresu pro komunikaci.

### 3.1.12 Mapování mezi adresou a ID uzlu

V systému bude existovat mechanismus pro mapování ID uzlu na adresu uzlu a naopak.

## 3.2 Architektura

Na základě zkušeností s předchozí verzí, zjištěných problémů a nově uvedených funkčních požadavků je možno navrhnout pro systém obecnou architekturu. Je nutno celou architekturu navrhovat, tak aby její rozšiřování o nové komponenty bylo zcela bez problémové. V rámci IoT se využívá celá paleta komunikačních modelů, které byly uvedeny v kapitole 2.1. Jako nejvýhodnější komunikační model se osvědčil Publish – Subscribe, který využívají i jiné IoT platformy.

Zvolené technologie mají zásadní vliv na výslednou architekturu. Pokud dojde k výběru technologie vhodné pro zamýšlený účel, dopad na výslednou architekturu bude velmi pozitivní s ohledem na minimalizaci složitosti a náročnosti implementace.

### 3.2.1 Volba technologií

Jako jádro celého systému bude sloužit middleware Apache Pulsar. Přechod od Apache Kafka k Apache Pulsar slibuje zásadní zlepšení managmentu a škálovatelnosti middleware komponenty pro výměnu zpráv mezi uzly. Apache Pulsar má na rozdíl od Apache Kafka brokery bezstavové. To znamená jednodušší horizontální škálování, které je možno provádět flexibilně dle zátěže, nové brokery se nemusejí s ostatními synchronizovat a jsou schopny se ihned zapojit do činnosti clusteru. Dalším důvodem pro jeho volbu je možnost vykonávání bezserverového kódu přímo v jednotlivých brokerech Apache Pulsar. Díky tomu není nutno vytvářet a nasazovat kompletní aplikaci s jednoduchou logikou pro zpracování dat a je možno kód logiky nasadit přímo do Pulsar clusteru. Výhodou je i fakt, že Apache Pulsar má nativní podporu pro Apache Kafka a tyto dvě technologie lze mezi sebou zaměnit. Posledním zásadním důvodem pro volbu Apache Pulsar je možnost instalace a tvorby adaptérů. Díky tomu lze Pulsar využívat například i jako MQTT brokera.

Pro výměnu zpráv byl zvolen serializační framework Apache Avro. Ten byl vybrán, protože je nezávislý na jazyku a zároveň má Apache Pulsar pro Avro nativní podporu v podobě registru schémat. Díky tomu mohou být omezeny chyby při deserializaci a zajištěna typová bezpečnost. Dále je serializace a deserializace pomocí Apache Avro velice rychlá a výstupní serializovaná data jsou minimalistická. Avro také umožňuje obousměrnou konverzi dat mezi JSON a Avro formátem, a možnost využít definované Avro objekty, resp. schémata, pro uložení dat do souboru. Možnost využít dynamické typování je výhodou usnadňující použití různých programovacích jazyků.

Celý systém je navrhován tak, aby bylo možno využít libovolný programovací jazyk. Lze očekávat, že dominantní jazyky v tomto systému budou Java a Python.

Pro nasazení jednotlivých komponent systému na základě funkčního požadavku 3.1.5 je zvolena technologie kontejnerů. Předpokládá se vývoj v Dockeru a následné produkční nasazení v Kubernetes clusteru. Oba systémy využívají stejný formát pro obrazy definovaný v rámci Open Container Initiative[24] a tedy aplikace, resp. obrazy kontejnerů jsou mezi nimi navzájem přenositelné. Tento formát je kompatibilní i s OpenShift, takže jeho využití nabízí dobrou přenositelnost mezi různými kontejnerovými systémy a více možností nasazení.

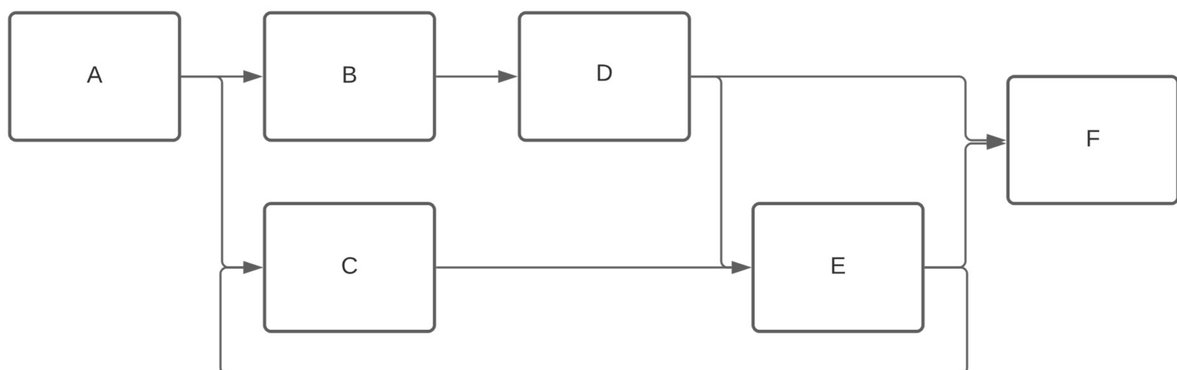
[24] https://opencontainers.org/

### 3.2.2 Obecná architektura platformy

IoT platforma musí být navržena tak, aby byla, pokud možno, eliminována centralizace. Výpadek jedné části by ideálně neměl ohrozit chod ostatních. S tímto problémem se lze setkat například u systému Node-RED, kde výpadek instance znamená výpadek všech funkcí systému. Důvodem je monolitická architektura aplikace.

Jako nejvhodnější řešení se jeví použití mikro-servisní architektury a komunikace pomocí publish-subscribe přes Apache Pulsar, který tento model komunikace nativně podporuje. V takovém případě je ale potřeba vyřešit tok dat. Ten je u běžných mikro-servisních aplikací řešen statickou konfigurací a přesně definovanými rozhraními pro komunikaci. V rámci IoT platformy je nutné data mezi jednotlivými uzly, resp. aplikacemi směrovat dynamicky.

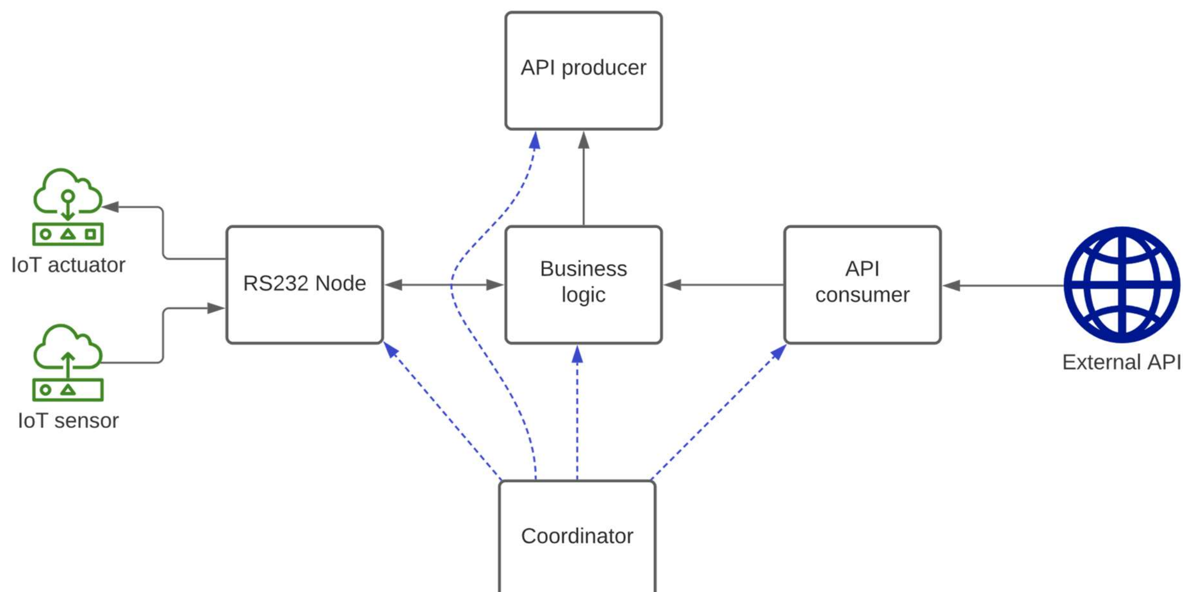
Pro splnění výše uvedeného se nabízí využít programovací paradigma FBP, kde se aplikace definuje jako síť černých skříněk (black-box), které mají své vstupy, výstupy a mezi sebou si předávají zprávy po předem definovaných spojeních. [27] Příklad FBP je zobrazen na obrázku 4. Každý uzel, resp. aplikace představuje jeden black-box. Tento přístup umožní propojovat jednotlivé uzly do komplexních řetězců, stromů či obecných orientovaných grafů, které budou vykonávat svou funkci. To má zásadní výhodu v tom, že pokud dojde k výpadku v jednom grafu, pak jiný nezávislý graf nebude zasažen. Další výhoda je vysoká znovupoužitelnost jednotlivých mikro-servisních komponent, které mohou být napsány pro jeden účel, ale využity v různých grafech. Například komponenta pro výpočet dlouhodobých průměrů nebo adaptér na RS232.




Obrázek 4 Příklad Flow-Based Programming

Propojování jednotlivých uzlů do těchto funkčních celků – grafů bude možné dvěma způsoby. První je manuální, kde jednotlivé uzly budou využívat svůj lokální konfigurační soubor. Druhý způsob je pomocí koordinátora, který konfiguraci uzlům předá vzdáleně. Díky

tomu bude možné funkční celky upravovat a rozšiřovat za běhu bez nutnosti manuálního zásahu na jednotlivých uzlech. Na obrázku 5 je zobrazen příklad jednoduché aplikace využívající uvedenou architekturu.




Obrázek 5 Příklad realizace v uvedené architektuře

### 3.2.3 Obecný model uzlu

Uzel sítě je nutno navrhnout velmi abstraktně tak, aby byla zajištěna možnost rozšiřitelnosti. Zároveň je ale třeba návrh omezit z důvodu zachování vysoké kompatibility mezi jednotlivými částmi. Při návrhu se lze inspirovat programovacím paradigmatem FBP.

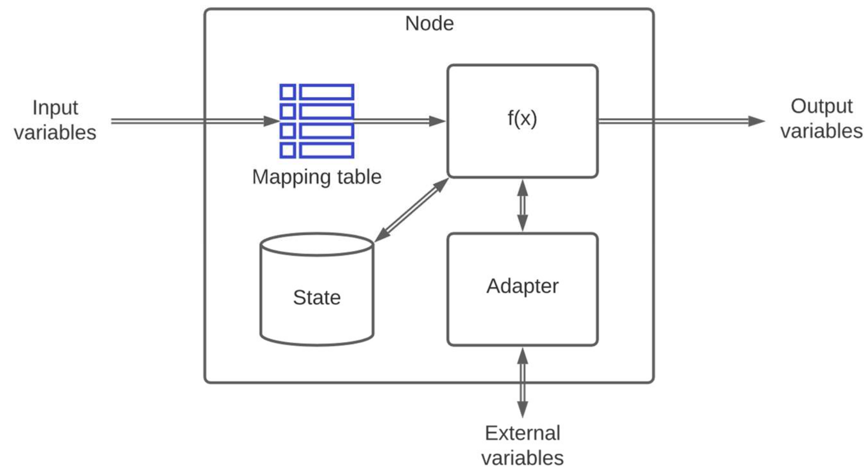
Při využití FBP bude mít uzel své vstupní a výstupní proměnné. Pomocí těchto proměnných bude komunikovat s ostatními uzly v síti. Dále bude mít navíc externí proměnné, které se nacházejí mimo síť a uzel s nimi pracuje pomocí adaptéru.

Uzel musí být schopen fungovat samostatně. Aby to bylo možné, musí mít vnitřní logiku a stav. Dále je nutné výstupní proměnné jednoho uzlu provázat se vstupními proměnnými jiného uzlu. Toho lze dosáhnout pomocí mapovací tabulky. Na obrázku 6 je nákres celkové architektury obecného uzlu.

Přicházející hodnoty ze sítě jsou nejdříve namapovány pomocí mapovací tabulky na vstupní proměnné. Tato tabulka může být definována externě, například koordinátorem, nebo přímo na uzlu pomocí konfiguračního souboru.

Dále se v uzlu nachází jeho business logika (v obrázku 6 označeno jako f(x)). Ta může být jakákoliv a obecně realizuje funkci uzlu. Může se jednat o zpracování dat, čtení a zápis vnitřního stavu, externích proměnných nebo publikaci výstupních proměnných do sítě.

Adaptér zajišťuje přístup k externím proměnným pro čtení a zápis. Tyto externí proměnné jsou například data čtená přes sériovou linku ze senzoru nebo data získaná přes REST API externí služby.




Obrázek 6 Obecný model uzlu

### 3.2.4 Adresace a pojmenování uzlů

Aby bylo možné komunikovat mezi uzly, je nutno vytvořit systém pro jejich adresaci a pojmenování.

Jméno uzlu slouží k jeho jednoznačné identifikaci napříč sítí a je perzistentní. Pod tímto jménem jsou uloženy jeho konfigurace, je veden v koordinátorovi nebo je zobrazen uživateli. Jméno může být libovolně dlouhé, ale je doporučeno, aby se skládalo pouze z alfanumerických znaků[25] bez znaků speciálních. Mělo by být člověkem čitelné a snadno zapamatovatelné.

Adresa uzlu slouží primárně pro zasílání zpráv na konkrétní instanci. Adresa je neperzistentní a uzel při každém restartu používá jinou. Toto opatření umožňuje detekovat restarty uzlu a kolize jmen. Pokud dojde k situaci, že se v síti vyskytují dva uzly se stejným jménem, lze je rozlišit pomocí adresy a kolizi vyřešit. Jako adresa musí být použit textový řetězec, u kterého je zajištěna unikátnost. Tento řetězec musí být vygenerován uzlem při jeho startu. Je proto doporučeno využít jako adresu Universally Unique Identifier – UUID[26] ve verzi

[25] Znaky z rozsahu 0-9, a-z, A-Z a symbol pomlčka „-“
[26] RFC-4122: https://www.ietf.org/rfc/rfc4122.txt

4, která je založena na pseudo-náhodných čísel. Příkladem UUID v4 je 674f7350-9251-423b-9c9d-03e038e3437f.

V případě kolize jmen uzel, který se připojil k síti, provede svou deaktivaci a do chodu sítě se nezapojuje.

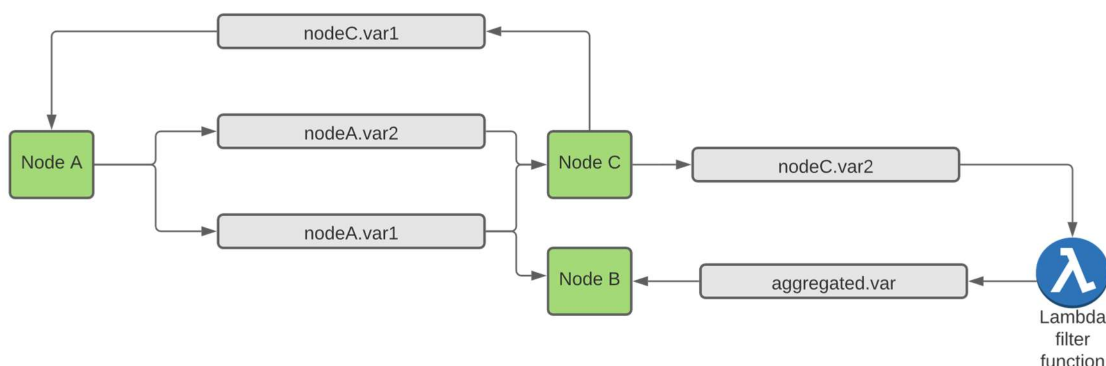
Provázání jména uzlu a adresy je nezbytné hlavně pro koordinátora a zobrazení uzlů pro uživatele. Toto provázání řeší objekt s popisem uzlu, resp. NodeDescriptor popsaný v kapitole 4.1.2.1. Ten je uzlem posílán v situacích popsaných blíže v rámci kapitoly zaměřené na implementaci.

### 3.2.5 Model komunikace

Uzly budou mezi sebou komunikovat pomocí témat v Pulsaru. Každá proměnná produkovaná uzlem bude mít vlastní téma, jehož název bude její identifikátor. Jako identifikátor proměnné se využije schéma, kdy je na začátku název uzlu, následuje tečka (.), za kterou bude název proměnné (např. uzel1.promenna1). Jednotlivé uzly se přihlásí k odběru svých vstupních proměnných a těch, které mají na své vstupní proměnné namapovány dle konfigurace. Pro své výstupní proměnné se uzly zaregistrují jako producenti.

Toto řešení umožňuje směrování zpráv mezi uzly tak, aby nebyly zasílány uzlům, kterým nepatří. Díky tomu je přenos dat velice efektivní a dochází k úspoře přenášeného objemu. Koncept je zobrazen na Obrázek 7 Příklad komunikace – výměna hodnot proměnných. Zelenou barvou jsou vyznačeny jednotlivé uzly, šedou jednotlivá témata. Také je zde zobrazen příklad využití bezserverového kódu, resp. lambda funkce nasazené v Pulsaru. Směr šipek označuje tok dat mezi uzly a tématy.

Pro servisní komunikaci mezi uzly bude využito podobného přístupu. Každý uzel se zaregistruje jako konzument tématu se svým identifikátorem. Tímto je vyřešena komunikace bod-bod. Dále je však nutno zajistit všeobecnou komunikaci pro koordinaci uzlů. Proto bude ještě existovat jedno broadcastové téma, které bude sloužit pro registraci nově přidaných uzlů.




**Obrázek 7 Příklad komunikace – výměna hodnot proměnných**

### 3.2.6 Koordinovaný a nekoordinovaný mód

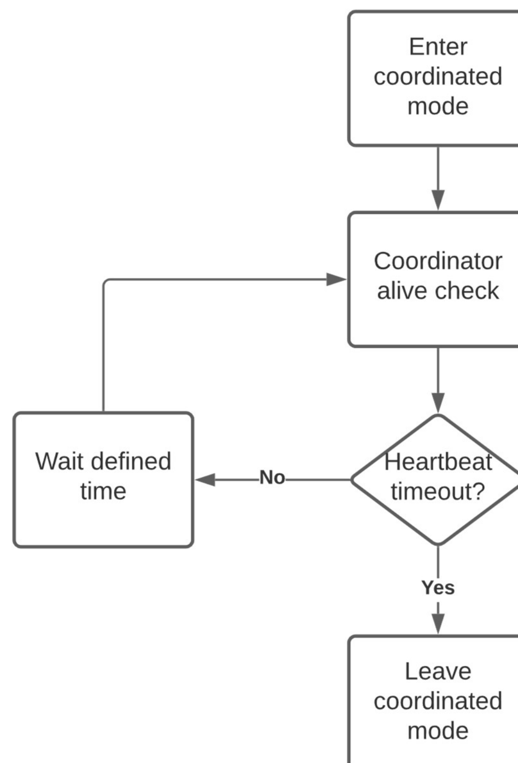
Platforma bude umožňovat provoz uzlů v koordinovaném a nekoordinovaném módu. To zajistí lehkou správu jak malých, tak i velkých aplikací.

V nekoordinovaném módu jednotlivé uzly pracují s lokální konfigurací mapovací tabulky. Konfigurace je uzlu předána manuálně při jeho nasazení v síti. Tento mód je určen pro použití v malých aplikacích, kde manuální správa je výrazně méně náročná než správa koordinátora. Tento mód taky najde uplatnění při vývoji a testování aplikací.

Koordinovaný mód znamená, že konfigurace jednotlivých uzlů je spravována pomocí koordinátora. To umožňuje spojovat uzly do orientovaných grafů z jednoho místa bez nutnosti manuálních změn v jejich konfiguracích. Koordinátor poskytuje jednotlivým uzlům konfiguraci jejich mapovací tabulky a převádí je mezi koordinovaným a nekoordinovaným módem. Uzel v koordinovaném módu zcela ignoruje svou lokální konfiguraci. Koordinovaný mód však může být využit pro aktualizaci lokální konfigurace uzlů, které běžně pracují v nekoordinovaném módu. To je provedeno tak, že koordinátor převede uzel do koordinovaného režimu a následně mu pošle novou konfiguraci spolu s příkazem na přepsání lokální konfigurace. Koordinovaný mód představuje dodatečnou úroveň zabezpečení, protože uzly, které jsou v tomto módu ignorují změny lokální konfigurace a zároveň nemohou být převedeny pod jiného koordinátora.

Proces přechodu mezi oběma módy je vždy inicializován koordinátorem, který uzlu posílá zprávu s příkazem na přechod do koordinovaného módu. Pokud uzel je v nekoordinovaném módu, pak příkaz potvrdí a přejde do koordinovaného pod daným koordinátorem. Pokud již uzel v koordinovaném módu je, příkaz odmítne.

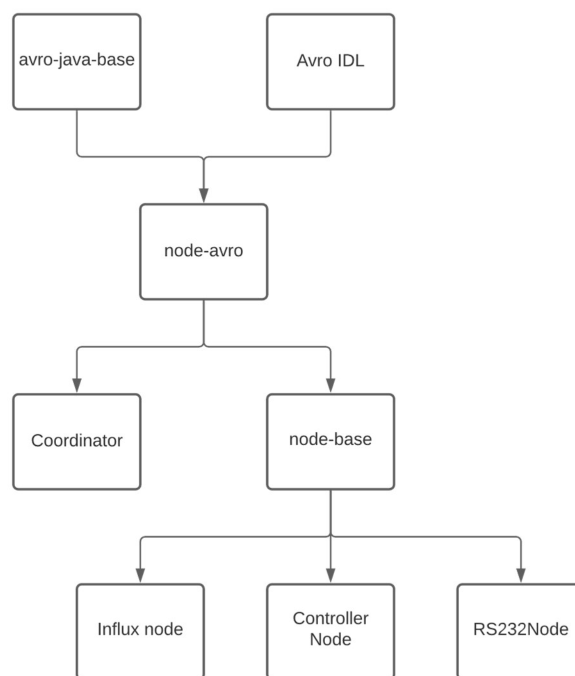
Je nutno vyřešit převod uzlů z jednoho koordinátora na druhý v případě servisních nebo havarijních situací, kdy je potřeba původního koordinátora odstavit a uzly převést pod jiného. Standartní postup je, že původní koordinátor své uzly přepne do nekoordinovaného módu a nový je převede pod sebe. Může však nastat situace, kdy původní koordinátor nečekaně vypadne a uzly neuvolní. V takovém případě uzly vyčkají předem definovanou dobu, než samy přejdou do nekoordinovaného módu. Tato doba je určena pro případný restart původního koordinátora. Koordinátor v pravidelném časovém intervalu posílá zprávu všem svým uzlům, že je stále aktivní. Mechanismus automatického přechodu uzlu do nekoordinovaného módu je zobrazen na Obrázek 8 Hlídání stavu koordinátora uzlem.




**Obrázek 8 Hlídání stavu koordinátora uzlem**

# 4 Implementace

Implementace Proof-of-Concept (PoC) je řešena pomocí jazyka Java a frameworku Spring Boot. Je rozdělena do pěti na sebe navazujících částí. Jako první je nadefinován komunikační protokol pomocí Avro IDL – AVDL. V druhé části je z této definice automatickým nástrojem vytvořen jar archiv, který obsahuje implementaci komunikačního protokolu v Javě. V rámci třetí části je implementováno hlavní jádro uzlu, které zajišťuje celkovou funkcionalitu. Čtvrtá část představuje jednoduchý příklad implementace konkrétního uzlu. Pátá část je implementace koordinátora. Závislost jednotlivých částí je vyobrazena na obrázku 9.




**Obrázek 9 Závislost částí projektu**

## 4.1 Komunikační protokol

Komunikační protokol je rozdělen na dvě části. První část řeší přenos hodnot proměnných. Druhá část zajišťuje komunikaci mezi uzly nebo mezi uzly a koordinátorem. Díky oddělení hodnot od komunikace mezi uzly je zajištěna možnost rozvoje celého systému a snadná integrace s jinými systémy.

### 4.1.1 Přenos hodnot

Přenos hodnot je vyřešen pomocí objektu Value, jenž má 4 atributy. Atribut typu string variableId označuje, ke které proměnné se daná hodnota váže, msTimestamp typu long je časové razítko vzniku dané hodnoty, valueType je enum uvádějící typ přenášené hodnoty a atribut value je samotná přenášená hodnota. Níže je zobrazen fragment Avro IDL kódu týkající se objektů pro přenos hodnot.

```
enum EValueType{
   BOOLEAN, INT, LONG, FLOAT, DOUBLE, BYTES, STRING
}


record Value{
   string variableId;
   long msTimestamp;
   EValueType valueType;
   union{boolean, int, long, float, double, bytes, string} value;
}
```

Toto řešení umožňuje přenášet jakoukoliv hodnotu. Přenos číselných hodnot je zcela triviální díky typům int, long, float a double. Ostatní nečíselné hodnoty mohou být přenášeny v serializované podobě jako string, například JSON, nebo jako bytes – pole bajtů. Maximální délka hodnoty typu string nebo bytes je specifikací Avro stanovena na $2^{64}$ bajtů, tedy 18.45 EB. Tato hodnota je dostatečně velká na to, aby nebyla omezujícím faktorem. Lze tedy přenášet velmi komplexní data nebo celé soubory. Uzel přijímající tyto hodnoty musí být schopen je zpracovat, tedy rozumět jejich kontextu.

### 4.1.2 Komunikace mezi uzly

Komunikace mezi uzly probíhá pomocí objektu NodeMessage. Ten obsahuje adresu odesílatele, adresu příjemce, časové razítko, typ zprávy a tělo zprávy. V některých případech je možné, aby tělo zprávy bylo prázdné, protože stačí informace o typu zprávy a není potřeba předávat dodatečná data. Z toho důvodu může být více typů zpráv, než je objektů přenášených v těle zprávy. Níže je zobrazen fragment Avro IDL kódu demonstrující jednotlivé typy zpráv a strukturu objektu NodeMessage.

```
enum EMessageType{
   REGISTRATION_REQUEST, REGISTRATION_REJECT, ADVERTISE, JOIN_COORDINATED,
LEAVE_COORDINATED, HEARTBEAT, CONFIG
}


record NodeMessage{
   string source;
   string destination;
   long msTimestamp;
   EMessageType messageType;
   union{null, NodeDescriptor, NodeConfig} payload;
}
```

#### 4.1.2.1 Zpráva s popisem uzlu – NodeDescriptor

Objekt typu NodeDescriptor obsahuje popis uzlu. Atributy jsou adresa uzlu, unikátní jméno uzlu a seznam proměnných, kterými uzel disponuje. Zprávu s tímto objektem uzel odesílá všem ostatním uzlům pomocí broadcastu při svém startu a slouží pro jeho registraci v síti. V tomto případě odesílající uzel nevyplňuje pole proměnných – vars, protože ostatní uzly tuto informaci nepotřebují a nezpracovávají. Tato zpráva je také uzlem odeslána koordinátorovi, pokud uzel přechází do koordinovaného režimu (kapitola 4.4.1). Zde je již pole vars vyplněno. Níže je zobrazen fragment Avro IDL kódu se strukturou NodeDescriptor objektu.

```
enum EVariableDirection{
   OUT, IN
}

record Variable{
   string id;
   string friendlyName;
   EVariableDirection direction;
   EValueType valueType;
}

record NodeDescriptor{
   string nodeAddress;
   string nodeName;
   array<Variable> vars;
}
```

#### 4.1.2.2 Zpráva s konfigurací uzlu – NodeConfig

NodeConfig je odesílán koordinátorem tomu uzlu, který je v koordinovaném režimu. Tento objekt definuje interval v milisekundách, v jakém se má uzel koordinátorovi hlásit – heartbeat, zda má být konfigurace uložena lokálně a mapovací tabulku proměnných. Níže je zobrazen fragment Avro IDL kódu se strukturou NodeConfig objektu.

```
record VariableMapping{
   string fromVarId;
   string toVarId;
}

record NodeConfig{
   long heartbeatInterval;
   boolean overwriteLocal;
   array<VariableMapping> mappings;
}
```

### 4.1.3 Vzájemná detekce uzlů a koordinátorů

Vzájemná detekce uzlů a koordinátorů uzlem je nezbytná pro správnou funkci celého systému. Koordinátor potřebuje vidět, které uzly jsou v síti připojené, aby je mohl v případě potřeby převést do koordinovaného režimu. Uzly musí vědět, že jsou v síti připojeni konkrétní koordinátoři z důvodu zajištění ochrany před vstupem neautorizovaného koordinátora.

Detekce koordinátorů uzlem probíhá na základě zpráv typu HEARTBEAT zasílaných koordinátorem do speciálního tématu. Pokud uzel poprvé detekuje koordinátora, uloží si jej a pošle mu zprávu typu ADVERTISE, ve které se nachází jeho kompletní NodeDescriptor, kde jsou vyplněny všechny položky. Uzel sleduje pravidelné HEARTBEAT zprávy od koordinátora. V případě, že je koordinátor přestane zasílat, uzel jej vyřadí ze seznamu aktivních koordinátorů.

Koordinátor detekuje uzly na základě zprávy ADVERTISE obsahující NodeDescriptor. Tu uzel koordinátorovi posílá ve dvou případech. V prvním případě dochází k odeslání této zprávy, pokud uzel přejde ze stavu Joining (4.4.1.1) do stavu Non-Coordinated (4.4.1.2). Druhý případ je, že uzel detekuje nově připojeného koordinátora a zároveň nachází ve stavu Non-Coordinated nebo Coordinated (4.4.1.3). Navíc je zaveden mechanismus pro sledování aktivity uzlů ve stavu Coordinated příslušným koordinátorem. Uzly v tomto stavu pravidelně posílají koordinátorovi zprávu typu HEARTBEAT.

### 4.1.4 Autorizace koordinátora

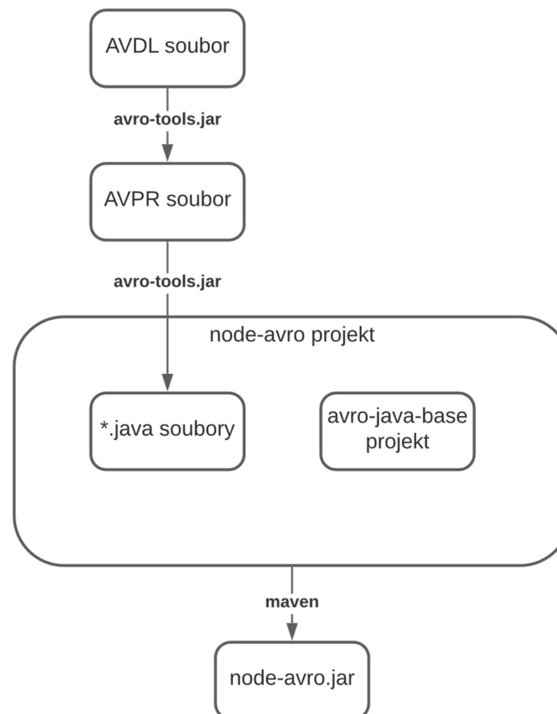
Pro zajištění ochrany sítě proti útoku v podobě zavedení neoprávněného koordinátora je využito speciálního tématu, kam mohou zapisovat pouze koordinátoři. Ti do tohoto tématu odesílají zprávy typu HEARTBEAT. Detaily autorizace koordinátorů jsou popsány v kapitole 4.6.

Koordinátor, který bude mít možnost do tohoto tématu zapsat, je považován za autorizovaného. Uzel od něj akceptuje zprávu typu JOIN_COORDINATED a přejde do koordinovaného režimu. Koordinátor, který nebude v tomto speciálním tématu propagován, není uzlem zařazen na seznam autorizovaných koordinátorů. V takovém případě uzel zapíše do logu pokus neautorizovaného koordinátora o převod do koordinovaného režimu a zprávu typu JOIN_COORDINATED ignoruje.

## 4.2 Převod Avro IDL na Java balík

Převod Avro IDL souboru (AVDL) na Java balík je řešen ve dvou částech. Jako základ je vytvořen java projekt s názvem avro-java-base. Tento projekt pouze obsahuje pom.xml a závislosti nutné k využívání Avro frameworku v Javě. Dále je vytvořen prázdný projekt node-avro, který obsahuje pouze pom.xml a jedinou závislost, a to avro-java-base. AVDL soubor je pomocí nástroje avro-tools zkompilován nejdříve na soubor Avro Protocol – AVPR ve formátu JSON. Ten je pomocí nástroje avro-tools znovu zkompilován a výstupem jsou Java třídy definované v AVDL souboru. Tyto třídy jsou poté vloženy do node-avro projektu, který je zkompilován nástrojem Maven a výstupem kompilace je Java balík obsahující kompletní implementaci Avro protokolu v Javě. Tento balík pak může být použit jako závislost v dalších projektech, které daný Avro protokol využívají. Celá operace je automatizována pomocí bash skriptu. Převod AVDL na Java balík je vyobrazen schematicky na obrázku 10.

Je nutno zmínit, že soubor AVPR je možno zkompilovat na soubory typu Avro Schema – AVSC ve formátu JSON, které jsou pak využitelné jinými jazyky, jako je například JavaScript a Python. Tím je zajištěn na jazyce nezávislý formát zpráv dle požadavku 1.4.3.

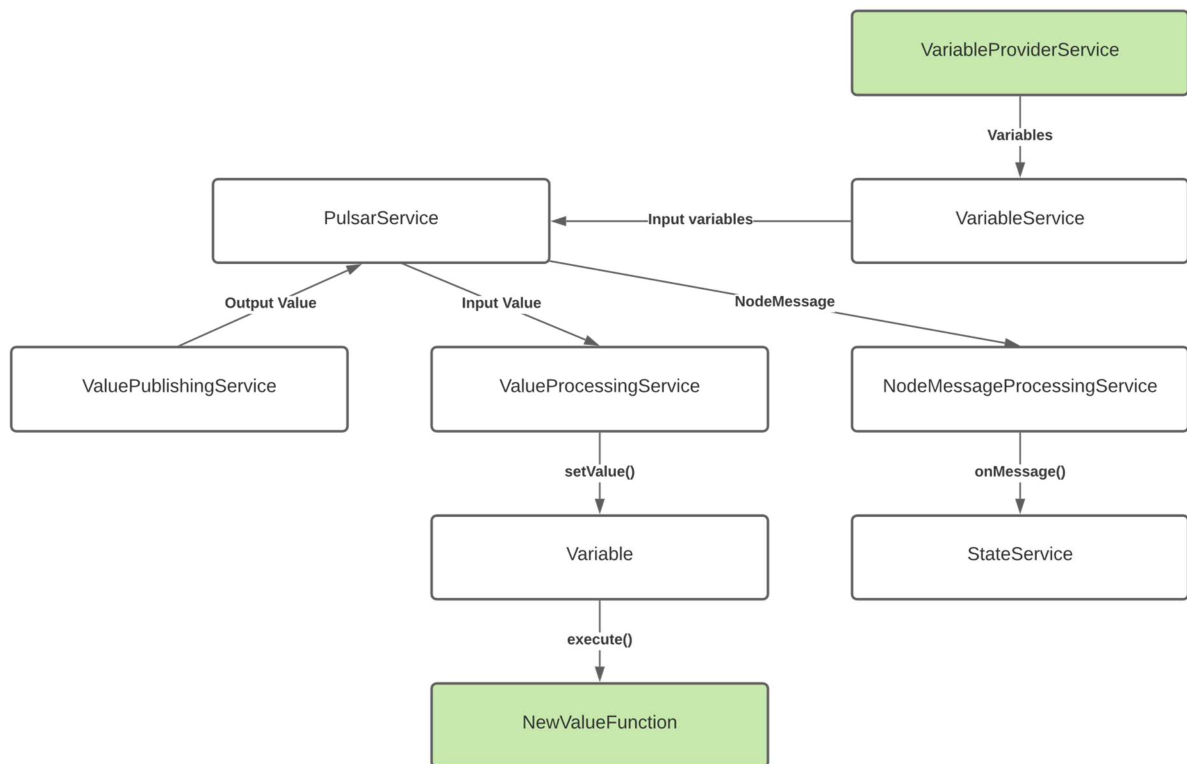



**Obrázek 10 Schéma převodu AVDL na jar balík**

## 4.3 Jádro uzlu

Jádro uzlu zajišťuje základní funkcionalitu uzlu sítě a je určeno pro implementaci konkrétních uzlů v jazyce Java. Řeší se zde napojení na Apache Pulsar, odesílání, příjem, zpracování hodnot a zpráv i řízení stavu uzlu. Jádro je založeno na frameworku Spring Boot ve verzi 2.5.6 a implementace uzlů je očekávána nad stejným frameworkem. Spring Boot byl zvolen pro jednoduchost a rychlost vývoje PoC verze. Ten má sice vyšší požadavky na výpočetní prostředky, ale není vyloučeno, že v budoucnu vznikne úspornější verze jádra bez využití frameworku. Bude určená pro zařízení s omezenými výpočetními prostředky, kde provoz aplikace založené na Spring Boot není možný.

Následující obrázek 11 zobrazuje přibližnou podobu jádra a způsob toku dat skrze jednotlivé služby. Zeleně vyznačené komponenty představují rozhraní, která musí být implementována při tvorbě konkrétního uzlu.




**Obrázek 11 Přibližná podoba jádra**

### 4.3.1 Služba pro správu proměnných

Správu proměnných zajišťuje služba VariableService. Ta poskytuje seznam identifikátorů konzumovaných proměnných pro PulsarService, inicializuje proměnné uzlu při jeho startu, umožňuje přístup k proměnným a spravuje mapovací tabulku. Aby bylo možné při startu uzlu načíst proměnné, VariableService vyžaduje existenci služby, která implementuje rozhraní VariableProviderService.

### 4.3.2 Služba komunikace

O napojení na Apache Pulsar se stará služba PulsarService. Ta zajišťuje odesílání a příjem dat (Value, NodeMessage), správu konzumentů a producentů. Konzumenti jsou vytvářeni a likvidováni na základě seznamu identifikátorů přijímaných proměnných. Vytváření a likvidace konzumentů probíhá vždy při změně mapovací tabulky. Pokud VariableService dostane nový seznam, dojde k likvidaci původního konzumenta a vytvoření nového. Konzument (třída Consumer) v Apache Pulsar podporuje odběr více témat, proto postačuje jeden objekt pro celý seznam.

Producenti jsou vytvářeni automaticky při odesílání dat dle identifikátoru proměnné – variableId v objektu Value, nebo identifikátoru cíle – destination v objektu NodeMessage.

Zde musí pro každý destinantion a variableId existovat separátní producent, protože funkce vícenásobného producenta není v rámci Apache Pulsar podporována.

### 4.3.3 Řízení stavu

Uzel je z definice stavový automat. Má jasně definované stavy a činnosti, které v nich má vykonávat. O tento úkol se stará služba StateService. Na základě stavu volá funkce dalších služeb a tím zajišťuje, že jsou pro každý stav vykonány příslušné činnosti.

### 4.3.4 Procesní služby

Dále jsou v jádře definovány služby pro zpracování Value a NodeMessage. V případě Value jsou to služby ValueProcessingService a ValuePublishingService. NodeMessage je zpracována ve službě NodeMessageProcessingService.

ValueProcessingService se stará o zpracování příchozích objektů Value, zajišťuje validaci, případnou konverzi datového typu a zápis hodnoty do příslušné vstupní proměnné. Validaci datového typu je nutno provádět z důvodu implementace struktury union v Javě. Union je v Javě reprezentován třídou Object. Proto je nutno ověřit, že příchozí objekt Value je očekávaného nebo kompatibilního datového typu.

ValuePublishingService zajišťuje odeslání hodnot nastavených do výstupních proměnných. Tato služba je automaticky napojena na všechny výstupní proměnné. Tím je zajištěno, že při zápise hodnoty do proměnné dojde k její propagaci do příslušného tématu v Pulsaru.

Zpracování všech příchozích zpráv typu NodeMessage je řešeno ve službě NodeMessageProcessingService. Tato služba zpracovává zprávy na základě tématu, ze kterého pocházejí. Zde je možno rozlišit, zda zpráva přišla jako broadcast, byla adresována přímo dotyčnému uzlu nebo pochází od autorizovaného koordinátora.

### 4.3.5 Implementace proměnných

Samotné proměnné jsou implementovány pomocí třídy Variable. Pro zajištění korektní inicializace instance Variable je k dispozici pomocná služba VariableFactory. Aby byla práce s proměnnými a jejich hodnotami jednoduchá, bylo definováno rozhraní NewValueFunction s jedinou metodou execute. To umožňuje do proměnné přidat dodatečnou logiku pro zpracování nové hodnoty. Je toho například využito u výstupních proměnných, kdy je do každé přidána instance ValuePublishingService, která novou hodnotu výstupní proměnné publikuje do příslušného tématu v Apache Pulsar.

## 4.4 Implementace uzlu

Implementace konkrétního uzlu může být minimalistická nebo komplexní. Záleží na množství a složitosti očekávaných funkcí. Práce s jádrem je však velmi jednoduchá. Stačí implementovat rozhraní VariableProviderService, přes které jsou poskytovány proměnné do jádra a rozhraní NewValueFunction, které umožňuje reagovat na nově příchozí hodnotu.

V rámci PoC jsou implementovány tři uzly. První řeší přemostění sériové linky a představuje tak příklad integrace existujícího hardware do sítě. Tento uzel nemusí datům proudícím po sériové lince rozumět, protože funguje jako pouhý adaptér. Druhý uzel umí data ze sériové linky interpretovat, obsahuje řídící logiku a data dekódovaná z bajtů publikuje dále jako hodnoty svých výstupních proměnných. Tento uzel obsahuje řídící logiku pro hardware. Funkce třetího uzlu spočívá v tom, že veškeré příchozí vstupní hodnoty ukládá do časové databáze InfluxDB. Je druhým příkladem univerzálního uzlu, který nemusí datům rozumět.

### 4.4.1 Stavový automat uzlu

Na základě návrhu v kapitole 3 se uzel může nacházet ve dvou stavech, nekoordinovaném a koordinovaném. Pro splnění požadavků 3.1.10 a 3.1.11 je nutno zavést další dva pomocné stavy, pro připojování uzlu – Joining a odmítnutí uzlu – Rejected. Tento stavový automat je na obrázku 12.

Stavový automat je implementován pomocí objektově orientovaného návrhového vzoru State Machine. Principem tohoto vzoru je, že existuje hlavní "obalová" třída, která obsahuje prostředky, které jsou využívány všemi stavy. Poté je použito buď společné rozhraní nebo abstraktní třída reprezentující předka pro každý stav. Jednotlivé stavy poté implementují dané rozhraní či rozšiřují abstraktní třídu a obsahují logiku pro daný stav. V tomto projektu je hlavní "obalová" třída pojmenována StateService, společný předek pro stavy je abstraktní třída State a jednotlivé stavy jsou jejími potomky. [28]

#### 4.4.1.1 Stav připojování – Joining

Tento stav je výchozí a slouží pro připojení uzlu k síti. Uzel odešle svůj deskriptor obsahující jeho ID všem ostatním uzlům v síti jako broadcast. Pokud má jiný uzel stejné ID, odešle novému uzlu jako odpověď zprávu typu REGISTRATION_REJECT. Nový uzel na tuto zprávu čeká 5 vteřin. Když v uvedeném časovém režimu zamítnutí neobdrží, přechází do nekoordinovaného stavu.

Časový limit 5 vteřin byl zvolen po pokusu, který ukázal, že největší možné zpoždění při doručování zpráv je cca 1200 ms. To je zhruba čtvrtina časového limitu, což by mělo představovat bezpečnou rezervu. Uvedené hodnoty jsou platné pro testovací prostředí a mohou se výrazně lišit v závislosti na provozních podmínkách. Postup k určení největšího zpoždění bude popsán v kapitole 5 zaměřené na testování a nasazení.

#### 4.4.1.2 Nekoordinovaný stav – Non-Coordinated

Uzel po přechodu do nekoordinovaného stavu odešle všem detekovaným koordinátorům zprávu se svým kompletním deskriptorem, která umožní každému koordinátorovi detekci nově připojeného uzlu pro následné přepnutí do koordinovaného režimu.

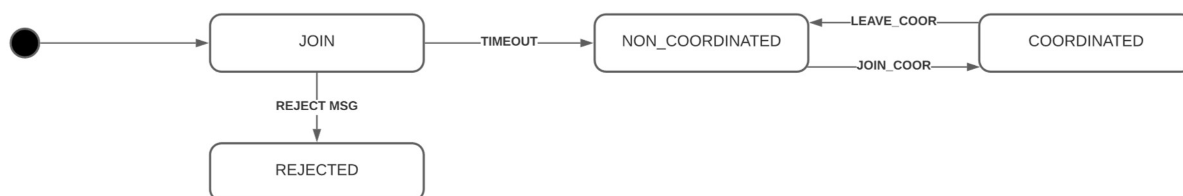
V nekoordinovaném režimu uzel pracuje dle své lokální konfigurace. Pouze v tomto stavu může být uzel přepnut do koordinovaného režimu a přejít pod koordinátora.

#### 4.4.1.3 Koordinovaný stav – Coordinated

Při převodu uzlu do koordinovaného stavu mohou nastat dvě situace. Pokud koordinátor disponuje konfigurací pro daný uzel, odešle mu ji hned při převodu a uzel tuto příchozí konfiguraci začne ihned používat. Jestliže však koordinátor žádnou konfigurací nedisponuje, odešle uzlu hodnotu null. Na základě null konfigurace zašle uzel koordinátorovi svou aktuální konfiguraci. Koordinátor pak může konfiguraci uložit, případně upravit a poslat ji zpět uzlu. Tímto je zajištěna konzistence konfigurace mezi uzlem a koordinátorem. Koordinátor může uzlu konfiguraci poslat jako perzistentní a neperzistentní. Perzistentní konfiguraci si uzel uloží jako lokální a po dalším startu ji použije jako výchozí v nekoordinovaném režimu. Neperzistentní konfiguraci uzel po svém restartu ztratí a použije původní lokální.

#### 4.4.1.4 Odmítnutý stav – Rejected

Pokud dojde ke kolizi a nový uzel má stejné ID jako jiný již běžící uzel, přejde nový uzel po obdržení zamítavé zpráv do odmítnutého stavu. V tomto stavu uzel na konzoli nebo do logu vypíše chybu, že byl odmítnut, a adresu uzlu, který jej odmítl. Jedná se o chybový stav, jenž lze napravit pouze změnou ID uzlu a jeho restartem. Z toho důvodu je stav Rejected konečný a uzel v něm je zcela neaktivní. Pouze informuje koordinátory pomocí zprávy typu ADVERTISE o tom, že se nachází ve stavu Rejected.




**Obrázek 12 Stavový automat uzlu**

### 4.4.2 Uzel UART (RS-232, RS-422, RS-485)

První implementovaný uzel je univerzální adaptér na sériovou linku Universal Asynchronous Receiver Transmitter – UART. Nad UART jsou provozovány standardy RS-232, RS-422 a RS-485. Tento uzel představuje příklad integrace systému s již existující technologií a hardwarem.

Zajištění přístupu k UART portům z Javy je vyřešeno použitím knihovny jSerialComm, která je na platformě nezávislá. Při použití správné konfigurace a doplnění příslušného hardware může být port UART spolu s knihovnou jSerialComm využit pro komunikaci po standardech RS-232, RS-422 a RS-485.

Pro každý UART port, který je na zařízení přístupný, jsou vytvořeny automaticky 3 proměnné – rx, tx a config. Proměnné rx a tx jsou typu bytes (pole bajtů) a reprezentují příslušné datové linky portu, rx je výstupní a tx je vstupní. Proměnná config je typu string a jedná se o vstupní proměnnou sloužící ke konfiguraci příslušného UART portu.

Logika tohoto uzlu je velmi jednoduchá. Nejprve musí být UART port nakonfigurován. To lze udělat zapsáním dat do proměnné config. Pokud je konfigurace úspěšná, je port otevřen pro čtení a zápis. Veškerá přijatá data na UART portu jsou odeslána jako hodnota příslušné proměnné rx do Pulsaru. Data přijatá z Pulsaru v proměnné tx jsou na příslušný UART port zapsána.

### 4.4.3 Uzel řídící logiky – kontrolér radiátoru

Druhý implementovaný uzel obsahuje řídící logiku – kontrolér pro radiátor. Umí zpracovávat konkrétní komunikační protokol z radiátoru, dekódovat data ze zpráv a dále je odesílat jako hodnoty svých výstupních proměnných.

Vzhledem k jednoduchosti logiky je implementace vyřešena pomocí tří tříd. První třída je konfigurační a zajišťuje vytvoření proměnných. Druhá je implementací rozhraní

VariableProvider a poskytuje proměnné do VariableService. Třetí je implementací rozhraní NewValueFunction obsahující logiku zpracování binárních dat a převod na hodnoty proměnných.

### 4.4.4 Uzel InfluxDB Sink

Třetí uzel slouží jako univerzální pro sběr dat a jeho úkolem je ukládat hodnoty vybraných proměnných do časové databáze InfluxDB. Identifikátory proměnných, které má tento uzel konzumovat jsou definovány vstupní mapovací tabulkou.

Tento uzel disponuje velmi jednoduchou logikou. Pro každý datový typ má jednu vstupní proměnnou. Samotná implementace je provedena, podobně jako u uzlu s řídící logikou, pomocí tří tříd. Také zde se nachází konfigurační třída zodpovědná za vytváření proměnných. Třída implementující rozhranní VariableProviderService je velmi podobná svou implementací stejné třídě v uzlu s řídící logikou. Třetí třída implementující rozhraní NewValueFunction obsahuje obsluhu databáze InfluxDB a zapisuje do ní příchozí hodnoty. Následují ukázky kódu implementací VariableProviderService a NewValueFunction, které tvoří hlavní část InfluxDB uzlu a demonstrují jednoduchost tvorby uzlu nad připraveným jádrem.

```
@Service
public class VariableProviderServiceImpl implements VariableProviderService
{
    private final List<Variable> variables;

    public VariableProviderServiceImpl(
            @Qualifier("booleanSinkVariable") Variable booleanSink,
            @Qualifier("intSinkVariable") Variable intSink,
            @Qualifier("longSinkVariable") Variable longSink,
            @Qualifier("floatSinkVariable") Variable floatSink,
            @Qualifier("doubleSinkVariable") Variable doubleSink,
            @Qualifier("bytesSinkVariable") Variable bytesSink,
            @Qualifier("stringSinkVariable") Variable stringSink
    ) {
        this.variables = List.of(
            booleanSink, intSink, longSink, floatSink,
            doubleSink, bytesSink, stringSink
        );
    }

    @Override
    public List<Variable> getVariables() {
        return this.variables;
    }
}
```

```
@Slf4j
@Service
public class InfluxDBService implements NewValueFunction {
    private final InfluxDBClient db;
    private final WriteApiBlocking writeApi;
    private final WritePrecision precision;

    private InfluxDBService(
            @Qualifier("influxDBUrl") String databaseURL,
            @Qualifier("influxDBToken") String influxToken,
            @Qualifier("influxDBOrg") String org,
            @Qualifier("influxDBBucket") String bucket
    ){
        this.db = InfluxDBClientFactory.create(
                databaseURL,
                influxToken.toCharArray(),
                org,
                bucket
        );
        this.db.setLogLevel(LogLevel.NONE);
        this.writeApi = this.db.getWriteApiBlocking();
        this.precision = WritePrecision.MS;
    }

    @Override
    public void execute(Variable var, Instant ts, Object val, String mvid){
        Point p = switch (var.getType()){
            case BOOLEAN -> toPoint(mvid, ts, (boolean) val);
            case INT -> toPoint(mvid, ts, (int) val);
            case LONG -> toPoint(mvid, ts, (long) val);
            case FLOAT -> toPoint(mvid, ts, (float) val);
            case DOUBLE -> toPoint(mvid, ts, (double) val);
            case BYTES -> toPoint(mvid, ts, (byte[]) val);
            case STRING -> toPoint(mvid, ts, (String) val);
        };

        this.writeApi.writePoint(p);
    }

    private Point toPoint(String varId, Instant ts, int val){
        return Point.measurement(varId)
                .time(ts.toEpochMilli(), this.precision)
                .addField("value", val);
    }

    //Zkráceno

}
```

## 4.5 Implementace koordinátora

Hlavním úkolem koordinátora je propojení uzlů do orientovaného grafu, čímž je zajištěn přenos hodnot mezi jednotlivými uzly. Pro tento úkol musí koordinátor umět komunikovat s uzly, poskytovat uživateli rozhraní pro práci s konfiguracemi a mít přístup k jejich úložišti. To znamená vytváření, upravování a mazání konfigurací jednotlivých uzlů. Protože se

koordinátor do samotného chodu sítě zapojuje jen za účelem konfigurace uzlů, lze ho chápat jako nástroj, pomocí kterého lze spravovat decentralizovaný systém.

Koordinátor je implementován stejně jako uzel v jazyce Java pomocí frameworku Spring Boot verze 2.5.6. Pro komunikaci s uzly je využit jako závislost předpřipravený balík node-avro. Uživatelské rozhraní je řešeno pomocí GraphQL a úložiště konfigurací uzlů je řešeno abstraktně pro lepší rozšiřitelnost systému.

### 4.5.1 Služba pro detekci uzlů – NodeService

Služba NodeService udržuje seznam aktivních uzlů v síti. Pokud koordinátor přijme deskriptor uzlu, je z deskriptoru vytvořen záznam o uzlu v této službě. Tím je umožněno získat informace o uzlu pomocí jeho adresy nebo ID. Tato služba zajišťuje překlad adres na ID, a naopak dle požadavku 3.1.12. Obecně probíhá komunikace mezi uzly pomocí neperzistentních unikátních adres a unikátní perzistentní ID představuje identifikátor dané instance. Tímto způsobem je možno detekovat restarty uzlů. Zároveň je unikátní perzistentní ID pro člověka lépe čitelné než náhodně generovaný UUID řetězec.

### 4.5.2 Služba pro správu konfigurací – NodeConfigStorage

Správa konfigurací uzlů je zajištěna pomocí služby NodeConfigStorage. Jedná se o rozhraní se dvěma funkcemi get a store. Obě operace využívají jako parametr unikátní ID uzlu. To umožňuje použít různé implementace, protože požadavky na ukládání konfigurací se mohou měnit.

V PoC jsou využity dva způsoby. Pro vývoj jsou konfigurace ukládány do operační paměti pomocí InMemoryNodeConfigStorage. V testovacím provozu PoC se přejde na uložení do souboru pomocí LocalAvroFileNodeConfigStorage. Konfigurace uzlů budou uloženy v lokálním souboru koordinátora. Tento soubor bude vytvořen pomocí frameworku Avro, který má podporu pro tvorbu Avro souborů na základě definovaného schématu. Jako ukládaný objekt se používá NodeConfig v binární serializované podobě. Soubory s konfigurací jsou pojmenovány dle uzlů, kterým náleží.

### 4.5.3 Služba pro koordinované uzly – CoordinatedNodeService

Služba zajišťuje převodu uzlů mezi koordinovaným a nekoordinovaným režimem, jejich monitoring a zasílání konfigurací. Jedná se o nástavbovou službu nad NodeService a NodeConfigStorage.

## 4.6 Zabezpečení

Důležitou součástí je i zabezpečení celého systému. Lze ho řešit ve dvou úrovních. Zabezpečení na úrovni jednotlivých uzlů – aplikací a zabezpečení přístupu ke komunikačnímu médiu – Apache Pulsar.

Systém je navrhován jako platforma určená k tvorbě IoT aplikací v enterprise prostředí s případnou integrací aplikací pro velká data – BigData a umělou inteligenci – AI. Každá aplikace integrovaná do této platformy může mít různé externí API a jinou bezpečnostní politiku. Z tohoto důvodu je značná část bezpečnostních opatření v kompetenci jednotlivých aplikací, které jsou do této platformy integrovány a jejich zabezpečení tedy není v této práci řešeno.

### 4.6.1 Zabezpečení Apache Pulsar

Zabezpečení přístupu ke komunikačnímu middleware – Apache Pulsar je základním prvkem bezpečnosti celého systému, protože přes Apache Pulsar probíhá veškerá komunikace. Apache Pulsar umožňuje řešit zabezpečení pomocí čtyř způsobů, a to TLS certifikátů, JSON Web Token – JWT[27] nebo systémů Kerberos[28] a Athenz[29].

Dalším prvkem zabezpečení Apache Pulsar je jeho multitenance. To znamená, že jednu instanci může využívat více entit – tenantů bez vzájemného ovlivňování. Pojmem tenant je myšlena například organizace. Multitenance je v Apache Pulsar řešena pomocí separace dat. Každý tenant může vytvářet a mazat jmenné prostory – namespace. Ty slouží k organizaci témat do skupin. Řízení přístupu a oprávnění je řešeno na úrovni jmenných prostorů, protože témata mohou vznikat a zanikat dynamicky.

Apache Pulsar realizuje autentifikaci a autorizaci ve dvou krocích. V prvním kroku je nutno definovat roli, resp. klienta. Není zde možnost vytvoření role, kterou pak lze přiřadit více klientům a pojmy "role" a "klient" jsou v rámci Apache Pulsar rovnocenné a navzájem zaměnitelné. Druhým krokem je přiřazení oprávnění jednotlivým rolím ke konzumaci a publikaci zpráv na úrovni jmenných prostorů.

Zabezpečení v rámci PoC musí řešit dva požadavky. Prvním je zajištění přístupu pouze pro uzly s validní autentifikací. Druhým je zajištění autorizace koordinátorů. To bylo nastíněno v kapitole 4.1.4.

[27] RFC-7519: https://datatracker.ietf.org/doc/html/rfc7519
[28] https://web.mit.edu/kerberos/
[29] https://www.athenz.io/

V rámci implementace PoC bylo pro zabezpečení zvoleno řešení pomocí JWT. Toto řešení bylo vybráno pro svou jednoduchost nasazení, správy a použití. Nasazení Kerberos nebo Athenz je pro potřeby PoC příliš náročné. Při porovnání mezi TLS a JWT je správa TLS certifikátů náročnější, protože vyžaduje vytvoření certifikační autority a párů veřejných a privátních klíčů. V případě JWT stačí vygenerovat pro každý uzel token pomocí tajného klíče a připraveného nástroje, následně nastavit příslušná oprávnění. Pro oddělení komunikace uzlů a koordinátorů byl využit jeden tenant a dva jmenné prostory (pozn.: dále prostor). První je prostor pro uzly s názvem "nodes" a druhý je pro koordinátory s názvem "coord".

#### 4.6.1.1 Zabezpečení uzlů

Pro každý uzel je vydán separátní JWT. Všechny uzly však mají zcela stejné oprávnění. Mohou v prostoru "nodes" konzumovat a publikovat zprávy a v prostoru "coord" zprávy pouze konzumovat. Tím, že uzel přijme zprávu od koordinátora, která pochází z tématu v prostoru "coord" má uzel zajištěno, že zpráva přišla od autorizovaného koordinátora. Teoreticky by šlo použít jeden token pro všechny uzly, ale v případě zneplatnění tokenu, by došlo k odpojení všech uzlů od sítě.

#### 4.6.1.2 Zabezpečení koordinátorů

V případě koordinátorů je řešení obdobné jako u uzlů. Jediný rozdíl je, že koordinátor může do prostoru "coord" zprávy publikovat.

# 5 Nasazení a testování

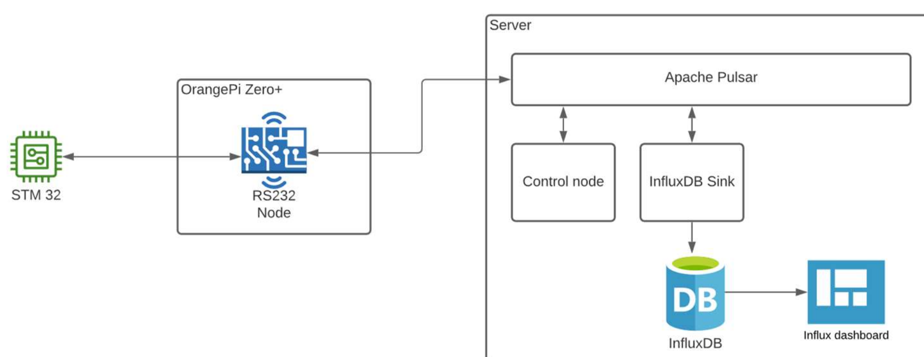
Nasazení a testování celého systému bylo prováděno v domácích podmínkách na modelovém příkladu řízení ventilátorů, které jsou namontovány na radiátoru. Celý systém je rozdělen do tří částí. Senzorová a aktuátorová část, uzel představující bránu, resp. adaptér a server, kde běží části, které nepotřebují pracovat s hardware.

## 5.1 Testovací prostředí

Na radiátor jsou umístěny dva analogové teplotní senzory LM35. Ty jsou přes převodník analog-digitál čteny pomocí vývojové desky s mikroprocesorem STM32. K tomuto mikroprocesoru je připojen na Pulse Width Modulation – PWM pin výkonový MOSFET tranzistor, přes který je řízena sada ventilátorů na spodní straně radiátoru. Mikroprocesor je pomocí seriové linky připojen k jednodeskovému počítači Orange Pi.

Orange Pi představuje bránu, resp. adaptér, který umožňuje připojení výše uvedeného zařízení se senzory a aktuátory do systému. Samotné Orange Pi je připojeno pomocí 100 Mbps ethernetové linky do lokální sítě.

V lokální síti se nachází domácí server, na kterém běží druhá instance uzlu obsahující logiku pro dekódování binárních dat odesílaných STM32 a řízení sady ventilátorů, instance databáze InfluxDB, uzel InfluxDB Sink a Apachce Pulsar. Obrázek 13 znázorňuje schéma nasazení PoC aplikace.




**Obrázek 13 Schéma nasazení**

### 5.1.1 Server

Pro kontext testování je nutné popsat server, kde je nasazena většina částí této práce. Procesor serveru je AMD Ryzen 5 5600G, disponuje 6 jádry a 12 vlákny. Operační paměť je 32 GB DDR4 3200 MHz. Datové úložiště je dvouúrovňové. První úroveň je PCI-E SSD disk, který slouží jako cache o velikosti 256 GB, kde jsou uložena data, která jsou často využívána.

Druhá úroveň je skupina pevných disků – HDD o souhrnné velikosti 7TB s paritní ochranou. Tyto disky slouží jako úložiště pro data, ke kterým není potřeba rychlý přístup a po většinu času jsou vypnuty. Docker kontejnery a případně virtuální stroje pracují nad SSD cache. Žádný kontejner v rámci této práce nemá uložená data v druhé úrovni a k využívání HDD tak nedochází.

Operační systém serveru je Unraid 6.9.2 a Docker je ve verzi 20.10.5. Unraid vychází ze systému Slackware a je optimalizován pro použití v datových úložištích a virtualizačních serverech.

### 5.1.2 Síťová infrastruktura

Většina sítě je v gigabitovém provedení a je využíváno virtuálních LAN – VLAN pro separaci provozu. Server, virtuální stroje a Docker kontejnery mají přidělenou vlastní VLAN. VLAN pro IoT zařízení je provozována v 100 megabitovém provedení na starším hardware.

## 5.2 Nasazení Apache Pulsar

Apache Pulsar je na serveru nasazen do Dockeru pomocí oficiálního standalone image, který již obsahuje Apache Zookeeper a Apache Bookeeper. Tato varianta nasazení je nejjednodušší, ale není schopná chodu v clusteru.

Původní záměr, použít pro testování variantu s clusterem, nebylo možno uskutečnit z důvodu chyby v oficiálním Docker image pro cluster variantu a nedostatečné dokumentace tohoto způsobu nasazení. Další dva způsoby, které byly v dokumentaci uvedeny, jsou nasazení pomocí instalace přímo na server a do Kubernetes clusteru. Oba způsoby se vyznačují značnou náročností, proto od nich bylo v rámci PoC upuštěno. Třetí možností nasazení Apache Pulsar v clusteru je do Docker Swarm[30] podpořeného clusterovým souborovým systémem GlusterFS[31]. Tato varianta sice není zmíněna v oficiální dokumentaci a Apache Pulsar neběží ve více instancích, ale jedná se také o validní zajištění vysoké dostupnosti.

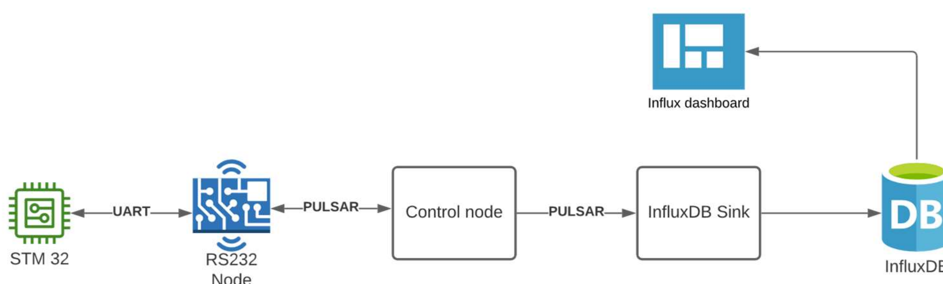
Z výše uvedeného je zřetelné, že Apache Pulsar v režimu vysoké dostupnosti lze nasadit různými způsoby s různou náročností, kdy každý způsob může být vhodný pro jinou situaci. Pro nasazení malého a lokálního charakteru může být validní nasazení jediné instance. V takovém případě se instalace na server nebo nasazení do Kubernetes clusteru nevyplatí kvůli

[30] https://docs.docker.com/engine/swarm/
[31] https://www.gluster.org/

náročnosti na práci a systémové prostředky. Pro kritické nasazení globálního charakteru lze však využít schopnosti Apache Pulsar vytvářet globální clustery z menších lokálních clusterů.

## 5.3 Nasazení uzlů

Pro demonstraci využití systému jako platformy jsou uzly nasazeny do jednoduchého řetězce tvořícího ucelenou aplikaci. Její logické schéma je zobrazeno na obrázku 14. Ta řídí ventilátory na radiátoru a data zobrazuje v jednoduchém uživatelském rozhraní. Samotná aplikace nedisponuje redundancí a ochranou proti selhání. V této situaci to není problém, protože v případě selhání jednoho ze dvou senzorů na radiátoru přestane celá aplikace plnit svou funkci. To představuje jediný bod selhání, který je možno odstranit pouze znásobením hardwaru. Redundanci řídícího uzlu, uzlu pro ukládání dat do InfluxDB nebo samotného InfluxDB lze zařídit pomocí orchestrace v Docker Swarm nebo Kubernetes.




**Obrázek 14 Logické schéma PoC aplikace**

### 5.3.1 Uzel UART (RS-232, RS-422, RS-485)

Jako zařízení pro běh programu je použit jednodeskový mini-počítač OrangePi Zero se čtyřjádrovým procesorem Allwinner H2+ a s 512 MB operační paměti (RAM). Jako operační systém byl použit Armbian ve verzi 21.08.1, což je minimalistická verze systému Debian určená pro jednodeskové počítače. Tento jednodeskový počítač byl zvolen záměrně z důvodu ověření funkčnosti implementace na levných ARM platformách s omezenými výpočetními prostředky.

Aplikace UART uzlu je nasazena jako služba operačního systému v podobě spustitelného jar archivu. Je zde nainstalován Java Runtime Enviroment – JRE ve verzi 17 a v rámci systemd je archiv zaregistrován jako služba. Je tak možno využívat příkaz systemctl pro její kontrolu a řízení.

### 5.3.2 Uzel řídící logiky – kontrolér

Uzel řídící logiky je nasazen na serveru jako Docker kontejner. Tento uzel nepotřebuje přístup k hardware, proto může být nasazen v kontejnerovém prostředí, což ulehčuje jeho správu a údržbu. V případě potřeby může být tento kontejner orchestrován pomocí Docker Swarm a tím lze zajistit jeho vysokou dostupnost.

### 5.3.3 Uzel InfluxDB Sink

Zde je situace obdobná jako u uzlu řídící logiky a nasazení je zcela shodné. Tento uzel pro svůj běh potřebuje instanci InfluxDB. Protože InfluxDB není klíčovou součástí navrhovaného systému a slouží jako příklad integrace systému s jinými službami tak není v této práci InfluxDB detailně rozebrán. Instance InfluxDB je nasazena také jako Docker kontejner na stejném serveru.

## 5.4 Testování

Testování systému bylo provedeno jako testování celku a bylo zaměřeno na oblasti využití systémových prostředků, objem komunikace, stabilitu a bezpečnost. Testování na vysokou dostupnost a eliminaci jediného bodu selhání bylo možné provést pouze v omezené míře z důvodu nedostatečných zdrojů.

### 5.4.1 Využití systémových prostředků

Sledování využití systémových prostředků bylo zaměřeno na operační paměť RAM a procesor. Určení využití operační paměti a procesoru je zcela zásadní pro odhad potřebné infrastruktury a jejího škálování. Data o využití HW prostředků byla sbírána dvěma způsoby z důvodu dvou různých nasazení uzlů, a to přímo na hardware (bare-metal) anebo jako Docker kontejner.

V případě bare-metal nasazení, které bylo využito pouze u UART uzlu, probíhal sběr dat pomocí linuxového příkazu free. Tento příkaz poskytuje informace o celkovém využití operační paměti. Protože na Orange Pi neběžela žádná jiná služba, bylo možno pomocí tohoto příkazu získat informace o využité operační paměti. Nejdříve se zjistilo, kolik operační paměti využívá samotný systém Armbian. Poté se nasadila aplikace UART uzlu, zjistilo se celkové využití paměti a odečetla se paměť využitá systémem. Tím se získalo přibližné využití paměti aplikací UART uzlu. Bylo zjištěno, že operační systém využívá 72 MB ± 5 MB paměti. Aplikace UART uzlu vyšla výpočtem na cca 160 MB operační paměti. Tato hodnota byla

potvrzena i pomocí monitoru systémových prostředků htop, který bylo zjištěno využití procesoru kolem 5 %.

Zjištit využití systémových prostředků v případě nasazení pomocí Docker kontejnerů je velmi jednoduché díky přímé podpoře této funkcionality Dockerem. Pomocí příkazu docker stats lze zobrazit aktuální využití procesoru a operační paměti jednotlivými kontejnery. Pro kontejner s řídící logikou bylo zaznamenáno využití paměti 150 MB a procesoru 0 %. Kontejner Influx Sink si vyžádal 180 MB paměti a 0 % procesoru.

Významné výchylky o více než 10 % ve využití operační paměti nebyly při týdenním testovacím provozu a pravidelném sběru dat zaznamenány. To je s velkou pravděpodobností zapříčiněno tím, že většina objektů je inicializována při startu aplikací a poté již nové objekty nevznikají. Tím nedochází k další alokaci operační paměti. Nízké využití procesoru je dáno nízkou frekvencí komunikace, řízením běhu programu pomocí událostí a vysokou výkonností procesoru AMD Ryzen 5 5600G.

Zjištěné hodnoty RAM se mohou jevit jako vysoké. To je dáno využitím frameworku Spring Boot v PoC. V současné době se množství dostupné RAM v zařízeních typu jednodeskových nebo průmyslových počítačů běžně pohybuje v řádech jednotek GB[32]. Jako minimální množství RAM, kterým musí použitý počítač disponovat, bylo určeno 512 MB. V případě použití menších hodnot RAM je možno jádro uzlu přepsat bez využití frameworku Spring Boot.

### 5.4.2 Objem komunikace

Zjištění objemu komunikace je nezbytné pro určení ideální kapacity konektivity. Systém navržený v předcházející bakalářské práci pro komunikaci využíval zprávy ve formátu JSON. Současný systém využívá serializaci do binárního formátu pomocí frameworku Apache Avro. To poskytuje významnou úsporu v objemu přenášených dat.

#### 5.4.2.1 Výpočet velikosti objektu Value

Nejčastěji přenášený objekt v rámci systému je objekt typu Value popsaný v kapitole 4.1.1. Dle dokumentace Apache Avro lze pro každý datový typ určit délku v bajtech. Datový typ string je serializován jako číslo typu long uvádějící délku stringu v bajtech, za kterým následuje daný počet bajtů. Stejným způsobem je serializován i datový typ bytes. Položky typu

[32] Pozorování založeno na dostupných jednodeskových počítačích pro roky 2020 a 2021, viz https://all3dp.com/1/single-board-computer-raspberry-pi-alternative/ a https://www.electronics-lab.com/top-10-single-board-computers-of-2020/

enum jsou serializovány jako integer uvádějící pořadí v enumeraci s indexací od 0. Celočíselné typy int a long jsou serializovány pomocí jednoho až čtyř, respektive osmi bajtů pomocí kódování s proměnnou délkou. Číselné typy s desetinnou čárkou float a double jsou serializovány jako 4 bajty, respektive 8. Logický typ boolean je serializován jako jeden bajt.

Na základě informací z předchozího odstavce lze určit následující vztah pro velikost objektu Value v bajtech – symbol $|V|$.

$$|V| = |ID| + |ts| + 2 + |v|$$

Symbol $|ID|$ označuje velikost identifikátoru proměnné – variableId, $|ts|$ označuje velikost časového razítka hodnoty – msTimestamp a $|v|$ označuje velikost samotné hodnoty – value. Číslo 2 je konstanta označující délku enumerace datového typu. Veškeré hodnoty jsou v bajtech.

Pro velikost $|ID|$ a obecně všech textových řetezců – stringů platí následující vztah, kde $|s|$ označuje počet bajtů serializovaného stringu a $s$ označuje počet bajtů ve stringu kódovaném v UTF-8. Toto kódování má proměnný počet bajtů na znak, proto je výpočet založen na počtu bajtů, nikoliv na počtu znaků. Horní celá část v součtu představuje počet bajtů, které budou využity pro zapsání počtu následujících bajtů stringu. Zde lze uvažovat $|ID| = |s|$. Stejný výpočet lze využít i pro datový typ bytes.

$$|s| = \left\lceil \frac{\log_2(s+1)}{7} \right\rceil + s$$

Velikost v bajtech datových typů int a long lze vypočítat podobným způsobem. Protože je využito kódování pomocí Zig-Zag, je výpočet rozdělen na dvě části. Část pro kladné hodnoty, značeno $|n^+|$, a část pro záporné hodnoty, značeno $|n^-|$. Hodnota 0 je kódována pomocí jednoho nulového bajtu. Zde $n$ označuje samotnou hodnotu int nebo long. Zde lze uvažovat $|ts| = |n^+|$.

$$|n^+| = \left\lceil \frac{\log_2(2n)}{7} \right\rceil \quad n > 0 \land n \in \mathbb{N}$$

$$|n^-| = \left\lceil \frac{\log_2(-2n-1)}{7} \right\rceil \quad n < 0 \land n \in \mathbb{N}$$

#### 5.4.2.2 Příklad a porovnání

Na konkrétním příkladu lze demonstrovat a porovnat velikost při přenosu dat pomocí Avro a JSON formátu. Pro porovnání bude použit příklad z reálného nasazení. Jde o přenos

šesti bajtů [0x80, 0x00, 0xFF, 0x20, 0x00, 0x18] s id proměnné “rs232node1.bytes.out“. Porovnání je provedeno mezi velikostí zprávy v předcházejícím systému z bakalářské práce a velikostí zprávy v systému navrženém v této diplomové práci.

V bakalářské práci by 6 bajtů bylo posláno ve formátu JSON jako Stavová zpráva rozhraní, kapitola 3.3.1.4 bakalářské práce, jejíž struktura je definována tabulkou 3.4. [1]

***Struktura stavové zprávy rozhraní, Zdroj: Generická IoT Síť, Jan Zubr, ČVUT FEL, 2019***

| ***Hodnota*** | ***Datový typ*** | ***Popis*** |
|---|---|---|
| *timestamp* | *Instant* | *Vznik zprávy* |
| *nodeLogin* | *String* | *Login fyzického uzlu* |
| *interfacesStates* | *Map <String,List<Integer>>* | *Mapa dat (identifikátor rozhraní => seznam bajtů)* |

Jako Stavová zpráva rozhraní by zpráva měla celkem 118 bajtů a její podoba je vyobrazena na obrázku 15. Žlutě zvýrazněné bajty představují variableId, modře označené msTimestamp a zeleně je vyznačena sekvence šesti přenášených bajtů.


```
{
  "nodeLogin": "rs232node1",
  "timestamp": "2019-01-21T05:47:26.853Z",
  "interfacesStates": {
    "1.1.0.0.1": [128,0,255,32,0,30]
  }
}
```


**Obrázek 15 Stavová zpráva rozhraní – příklad**

Při využití objektu Value a frameworku Avro bude mít zpráva celkem 36 bajtů. Obrázek 16 zobrazuje serializovaný objekt typu Value. Barevné označení je zde shodné, navíc je fialově označený datový typ. Bajty bez zvýraznění jsou servisní bajty protokolu Avro označující počty následujících bajtů.


```
15 72 73 32 33 32 6e 6f 64 65 31 2e 62 79 74 65 73 2e 6f 75 74 06 81 b2 c3
f4 a5 36 02 06 80 00 ff 20 00 18
```


**Obrázek 16 Serializovaný objekt Value**

Z pouhého vizuálního porovnání plyne, že formát Avro oproti formátu JSON přenáší data výrazně efektivněji a na uvedeném příkladě bylo možné stejnou informaci zakódovat do 36 bajtů místo původních 118. V sekvenci 36 bajtů jsou pouze 3 servisní bajty.

#### 5.4.2.3 Naměřená data

Měření přenesených dat probíhalo na dvou místech po dobu zhruba 6 dní. První měřící místo bylo nastaveno v hlavním síťovém routeru, který pomocí funkce Deep Packet Inspection sledoval přenos dat na Orange Pi. Druhé měřící místo byl Apache Pulsar, kde pro každého připojeného klienta dochází k automatickému záznamu přenesených bajtů.

Orange Pi odesílá každou sekundu jeden objekt Value se šesti bajty obdobně jak, bylo uvedeno v příkladu. Data z routeru naměřená pomocí nástroje Deep Packet Inspection ukázala, že za 6 dní bylo přeneseno cca 21,43 MB ve směru z Orange Pi. Tyto hodnoty nejsou zcela přesné, protože nástroj v routeru neumožňuje pokročilejší filtraci dat a odstranění vlivu komunikace servisních síťových protokolů NTP, ARP, DHCP, ping nebo SSH spojení a dalších. Podobná hodnota jako v routeru byla naměřena i na Apache Pulsar, kde bylo zaznamenáno 20,741 MB dat. To odpovídá zhruba hodnotě, kterou lze získat vynásobením počtu sekund v 6 dnech a 36 bajtů, což je cca 19 MB.

Naměřená data jsou dostatečně přesná pro porovnání s výsledky bakalářské práce, přestože se nejedná o zcela přesné měření. Bakalářská práce uvádí následující naměřené výsledky: *Původní řešení vzniklé v rámci semestrálního projektu přenese 3 GB dat za 48 hodin. Nyní se přenese pouze 0.15 GB dat za 48 hodin při stejné frekvenci a délce přenášených dat.* [1] Použití jiné technologie přenosu zcela jasně vedlo k úspoře dat. Za trojnásobnou dobu bylo přeneseno pouhých 13 % (20 MB z 150 MB) původního datového objemu. Při korekci na stejné časové období 48 hodin přenos v novém systému vychází na 6,2 MB. Jedná se tedy o cca 96 % nižší datový objem při přenosu stejné informace za stejný časový úsek. Tabulka 5.1 ukazuje souhrnný přehled objemu přenesených dat jednotlivých řešení.

**Tabulka 5.1 Porovnání řešení**

| | Objem dat za 48 h [MB] | Úspora [%] |
|---|---:|---:|
| **Semestrální projekt** | 3000 | 0,00 |
| **Bakalářská práce** | 150 | 95,00 |
| **Diplomová práce** | 6,2 | 99,79 |

### 5.4.3 Stabilita

Cílem testování stability systému bylo zjistit chování a nalézt případné chyby v situacích, kdy je hardware plně zatížen. Tyto testy byly zaměřeny primárně na server, protože je na něm nasazena většina komponent PoC, zejména Apache Pulsar. Zátěže serveru bylo dosaženo kombinací síťových přenosů na server z více zdrojů v podobě kopírování velkých souborů o velikostech více než 2 GB přes protokol Samba. Zároveň na byl serveru spuštěn syntetický stres test mprime pro zatížení CPU, vytvořen virtuální stroj, který obsadil 28 GB RAM. Proti Apache Pulsar byl spuštěn jednoduchý program, který produkoval a konzumoval zprávy vysokou rychlostí. Bylo sledováno, zda PoC aplikace dokáže v těchto podmínkách i nadále fungovat a zda dojde k neočekávanému chování.

Jediný problém, jenž se vyskytl, byl pád virtuálního stroje, který měl za úkol obsadit RAM. Podle logu operační systém serveru potřeboval uvolnit operační paměť, a proto ukončil proces s nejnižší aktivitou. PoC aplikace dokázala fungovat pouze s občasným zpožděním při doručování zpráv. Největší zaznamenané zpoždění bylo 1200 ms. Při běžném provozu bylo zaznamenáno dopravní zpoždění v rozsahu od 20 do 50 ms.

Dalším bodem ve sledování stability PoC aplikace byl její nepřerušený běh po dobu více než 6 dní. Každý den proběhla kontrola výskytu chyb a výjimek v logu. Náhodně byli také restartovány veškeré kontejnery a bylo sledováno, zda se systém vrátí zpět do funkčního stavu. To se stalo vždy, a tudíž nebylo třeba žádného zásahu. V případě výpadku Apache Pulsar držely jednotlivé uzly v paměti zprávy a po opětovném navázání spojení s Pulsarem tyto zprávy odeslaly.

### 5.4.4 Bezpečnost

V rámci testování bezpečnosti byly ověřovány 3 scénáře. Prvním scénářem bylo připojení uzlu bez oprávnění k Apache Pulsar, druhým ověření funkce autorizace, zda má uzel přístup ke čtení a zápisu pouze přidělených jmenných prostorů a třetí scénář byl stejný jako druhý, pouze z pozice koordinátora.

#### 5.4.4.1 Test přístupu k Apache Pulsar – Autentifikace

Test autentifikace byl proveden velmi jednoduše. Byl nastartován uzel bez tokenu, který se pokusil připojit k Apache Pulsar. Apache Pulsar odmítl spojení a uzel nebyl schopen číst ani publikovat do vymezených jmenných prostorů.

#### 5.4.4.2 Test autorizace uzlu

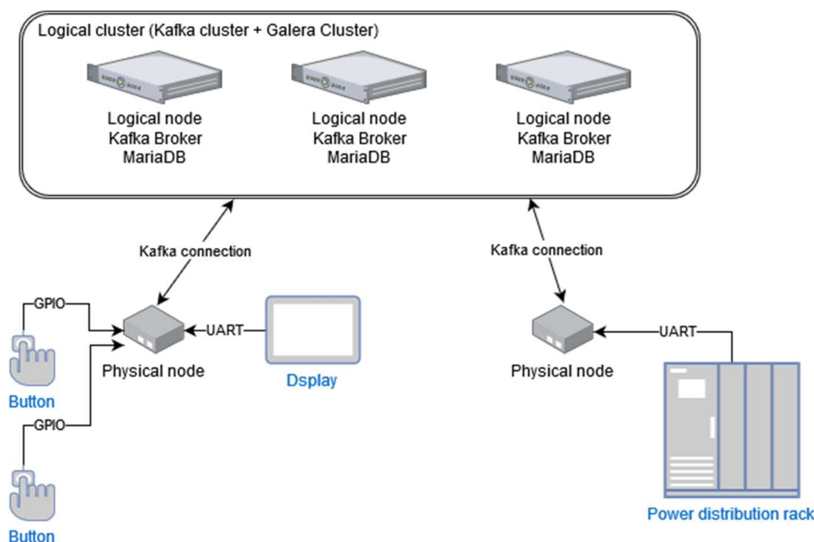
V rámci testu autorizace bylo zjišťováno, zda uzel dokáže číst oba jmenné prostory, tedy prostor pro uzly a koordinátory. Dále bylo otestováno, že uzel může zapisovat pouze do jmenného prostoru pro uzly, nikoliv pro koordinátory. Oba testy dopadly dle očekávání a uzel byl schopen oba prostory číst, ale zapisovat pouze do prostoru pro uzly.

#### 5.4.4.3 Test autorizace koordinátora

Testování autorizace koordinátora probíhalo stejně jako test autorizace uzlu. Jediný rozdíl byl v tom, že se testovalo, zda koordinátor dokáže zapsat do prostoru pro koordinátory. Tento test dopadl dle specifikace v kapitole 4.6.1.2 a vše fungovalo.

### 5.4.5 Porovnání modularity

Systém v předcházející práci byl navržen jako monolitický. Redundance bylo dosaženo replikováním Kafka brokeru, databáze MariaDB a aplikace logického uzlu. Architektura tohoto systému je zobrazena na obrázku 17, architektura aplikace logického uzlu byla ukázána na obrázku 1. Z uvedeného vyplývá, že modularita je velmi nízká a rozšíření či změna logiky velmi obtížná. Zároveň dochází ke zbytečné centralizaci, kdy nesouvisející části logiky se mohou navzájem ovlivnit.




**Obrázek 17 Architektura systému z bakalářské práce**

Nově navržené řešení využívá odlišnou architekturu založenou na volném propojení malých nezávislých služeb a je inspirováno FBP. Tím je velmi zjednodušena celková administrace systému, rozšiřování logiky a integrace s jinými systémy. Dále je možné dosáhnout decentralizace a redundanci řešit cíleně v těch částech systému, kde je to nezbytné.

# 6 Závěr

Cílem této práce bylo navázat na předcházející bakalářskou práci, rozšířit ji a vyřešit problémy odhalené během provozu původně navrženého systému. Požadavky na samotný IoT systém zůstaly téměř shodné. Byl kladen důraz na efektivitu, integraci s jinými systémy, zabezpečení a eliminaci jediného bodu selhání. Novým požadavkem bylo zaměření na enterprise prostředí a řešení systému jako platformy.

Před samotným návrhem nového systému bylo třeba provést analýzu existujících systémových řešení a technologií s ohledem na nové požadavky. Analýza byla zaměřena na komunikační modely, modely dle umístění logiky, middleware pro komunikaci a existující komerční IoT platformy.

Na základě analýzy a požadavků byla provedena volba technologií a zpracován návrh nového systému. V rámci návrhu byl kladen důraz na jednoduchost, modularitu a nezávislost na programovacím jazyce. Proto bylo využito mikro-servisní architektury. Celý návrh je inspirován programovacím paradigmatem FBP. Základním prvkem systému je program označovaný jako uzel. Každý uzel má své vstupní a výstupní proměnné. Jednotlivé uzly lze mezi sebou spojovat do orientovaných grafů – sítí propojením výstupních a vstupních proměnných. Takové sítě však mohou být velmi rozsáhlé a správa konfigurací jednotlivých uzlů náročná. Z toho důvodu byl vytvořen speciální uzel – koordinátor, který poskytuje možnost správy uzlů na jednom místě a jejich propojení do sítě. Jedná se spíše o nástroj, protože se na chodu samotné sítě aktivně nepodílí.

Pro demonstraci návrhu v reálném provozu byla vytvořena PoC aplikace. K její implementaci bylo využito jazyku Java a frameworku Spring Boot. Implementace byla rozdělena do několika částí tak, aby byla zajištěna znovupoužitelnost kódu. To mělo za následek velmi snadnou tvorbu jednotlivých uzlů, protože většina kódu byla mezi uzly sdílená. Díky tomu implementace uzlů spočívala pouze v implementaci jejich logiky. Koordinátor s uzly sdílí pouze kód týkající se komunikace, protože jeho funkce se od uzlů výrazně liší.

Implementovaná PoC aplikace byla nasazena do testovacího provozu a bylo ověřeno, že navržený systém plní požadovanou funkci a lze jej využít jako platformu pro tvorbu enterprise IoT aplikací. V rámci testovacího provozu se sledoval také objem přenesených dat. To je zásadní aspekt pro IoT systémy, protože v některých případech může být přenos dat omezen datovým limitem. Také byl proveden test stability systému pomocí stres testu

hardwarové infrastruktury. Zde systém také obstál a dokázal plnit svou funkci, přestože byla pozorována degradace ve výkonnosti, zejména v dopravním zpoždění.

V rámci této práce se podařilo splnit stanovené cíle a vyřešit problémy objevené při dlouhodobém provozu předcházejícího systému. Nový systém je vhodný pro použití ve velkých enterprise nasazeních, ale lze jej použít i pro malá nasazení, jako je například chytrý dům. Výsledek práce představuje velmi dobrý základ pro další rozvoj a tvorbu enterprise IoT aplikací. To je možné zejména díky FBP přístupu, použití Apache Pulsar a dobře navržené architektuře jádra uzlu. Ta dovoluje velmi snadný a rychlý vývoj aplikací nad společným kódem.

Vytvořený systém bude do budoucna dále rozšiřován a využíván. A to jak soukromě, tak i komerčně pro připravovanou platformu zaměřenou na IoT, BigData a umělou inteligenci, kde zajistí integraci na průmyslové protokoly a transformaci dat do jednotného formátu. Dále bude jádro uzlu přepsáno do varianty bez frameworku Spring. Tím dojde ke snížení požadavků na systémové prostředky.

# Seznam použité literatury

1. Zubr, Jan. *Generická IoT síť.* Praha : ČVUT FEL, 2019.

2. Coulouris G., Dollimore J., Kindberg T., Blair G. *Distributed Systems: Concepts and Design (5th Edition).* Boston : Addison-Wesley, 2011. 0132143011.

3. Sommerville, Ian. *SOFTWARE ENGINEERING, 9th Edition.* London : Pearson, 2011. 978-0-13-703515-1.

4. Hamilton, Eric. What is Edge Computing: The Network Edge Explained. *Web Cloudwards.* [Online] 27. Prosinec 2018. [Citace: 12. Říjen 2021.] https://www.cloudwards.net/what-is-edge-computing/.

5. Erl, Thomas. *Cloud Computing: Concepts, Technology & Architecture.* London : Pearson, 2013. 978-0133387520.

6. Which protocols does RabbitMQ support? *Web RabbitMQ.* [Online] 2021. [Citace: 15. 04 2021.] https://www.rabbitmq.com/protocols.html.

7. Routing. *Web RabbitMQ.* [Online] 2021. [Citace: 15. 04 2021.] https://www.rabbitmq.com/tutorials/tutorial-four-python.html.

8. Snyder, Bruce. *ActiveMQ in Action.* New York : Manning, 2011. 978-1933988948.

9. Pricing. *Web KubeMQ.* [Online] 2019. [Citace: 10. Říjen 2021.] https://kubemq.io/product-pricing/.

10. Introduction. *Web KubeMQ.* [Online] 2019. [Citace: 10. Říjen 2021.] https://docs.kubemq.io/.

11. Raje, Sanika. Performance Comparison of Message Queue Methods. *Digital Scholarship@UNLV.* [Online] Srpen 2019. [Citace: 10. Říjen 2021.] https://digitalscholarship.unlv.edu/thesesdissertations/3746/. 3746.

12. Documentation. *Web Apache Kafka.* [Online] 2021. [Citace: 10. Říjen 2021.] https://kafka.apache.org/documentation/.

13. Pulsar Overview. *Web Apache Pulsar.* [Online] 2021. [Citace: 10. Říjen 2021.] https://pulsar.apache.org/docs/en/concepts-overview/.

14. Avro Documentation. *Web Apache Avro.* [Online] 29. Říjen 2021. [Citace: 15. Listopad 2021.] https://avro.apache.org/docs/current/.

15. Home. *Apache Thrift.* [Online] 2021. [Citace: 10. Listopad 2021.] https://thrift.apache.org/.

16. Google. Protocol Buffers. *Google Developers.* [Online] 2021. [Citace: 10. Listopad 2021.] https://developers.google.com/protocol-buffers.

17. Documentation. *Web Node-RED.* [Online] 2021. [Citace: 15. Listopad 2021.] https://nodered.org/docs/.

18. ThingsBoard Documentation. *Web Thingsboard.io.* [Online] 2021. [Citace: 16. Listopad 2021.] https://thingsboard.io/docs/.

19. AWS IoT. *Amazon AWS.* [Online] 2021. [Citace: 15. Listopad 2021.] https://aws.amazon.com/iot/.

20. Azure IoT - Internet of Things Platform. *Web Microsoft Azure.* [Online] 2021. [Citace: 12. Listopad 2021.] https://azure.microsoft.com/en-us/overview/iot/.

21. Google Cloud IoT - Fully managed IoT Services. *Google Cloud.* [Online] 2021. [Citace: 15. Listopad 2021.] https://cloud.google.com/solutions/iot.

22. Internet of Things (IoT) - Cisco Jasper Control Center - Cisco. *Cisco.* [Online] 2021. [Citace: 10. Listopad 2021.] https://www.cisco.com/c/en/us/solutions/internet-of-things/iot-control-center.html.

23. Internet of Things | IBM. *Web IBM.* [Online] 2021. [Citace: 15. Říjen 2021.] https://www.ibm.com/cloud/internet-of-things.

24. An Introduction to IOTA. *IOTA Wiki.* [Online] 2021. [Citace: 10. Říjen 2021.] https://wiki.iota.org/learn/about-iota/an-introduction-to-iota.

25. Helium Documantation. *Web Helium.* [Online] 2021. [Citace: 10. Říjen 2021.] https://docs.helium.com/.

26. White Paper. *IoTeX: Building the connected worls.* [Online] 2021. [Citace: 10. Říjen 2021.] https://iotex.io/white-paper.

27. Morrison, J. Paul. *Flow-Based Programming, 2nd Edition: A New Approach to Application Development.* Scotts Valley, California : CreateSpace Independent Publishing Platform, 2010. 978-1451542325.

28. Shvets, Alexander. SourceMaking: Observer Design Pattern. *SourceMaking.* [Online] 2007 - 2019. [Citace: 20. Listopad 2021.] https://sourcemaking.com/design_patterns/state.

# Seznam zkratek

| Zkratka | Význam |
|---|---|
| AMPQ | Advanced Message Queuing Protocol |
| API | Aplication Programming Interface |
| AVDL | Avro IDL |
| AVPR | Avro Protocol |
| AVSC | Avro Schema |
| AWS | Amazon Web Services |
| FBP | Flow Based Programming |
| HTTP / HTTPS | Hypertext Transfer Protocol / Hypertext Transfer Protocol Secure |
| IaaS | Infrastructure as a Service |
| IDL | Interface Description Language |
| IoT | Internet Of Things |
| JMS | Java Messaging Service |
| JSON | JavaScript Object Notation |
| MQTT | Message Queuing Telemetry Transport |
| NAT | Network Address Translation |
| PaaS | Platform as a Service |
| PWM | Pulse Width Modulation |
| RAM | Random Access Memory |
| REST | Representational State Transfer |
| RFC | Request For Comment |
| RPC | Remote Procedure Call |
| SaaS | Software as a Service |
| SPOF | Single point of failure |
| SSL / TLS | Secure Sockets Layer / Transport Layer Security |
| STOMP | Simple / Streaming Text Oriented Message Protocol |
| USART / UART | Universal Synchronous / Asynchronous Receiver and Transmitter |
| USB | Universal Serial Bus |

# Příloha A: Seznam elektronických příloh

Součástí práce je elektronická příloha ve formě zip souboru. Ten obsahuje následující složky

- dp_avro-main – Avro definice, základní avro projekt a avro_tools nástroj
- dp_node-main – Hlavní jádro uzlu
- dp_rs232-node-main – Implementace uzlu UART / RS-232
- dp_radctrl-node-main – Implementace uzlu s řídící logikou pro radiátor
- dp_influx-node-main – Implementace InfluxDB uzlu
- dp_coordinator-main – Implementace koordinátora