



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ

DEPARTMENT OF INFORMATION SYSTEMS

SYSTÉM PRO ZPRACOVÁNÍ DAT Z CHYTRÝCH ZAŘÍZENÍ

SYSTEM FOR PROCESSING DATA FROM SMART DEVICES

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

Michal Bureš

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. Jiří Hynek, Ph.D.

BRNO 2024

Zadání bakalářské práce



153937

Ústav: Ústav informačních systémů (UIFS)
Student: **Bureš Michal**
Program: Informační technologie
Název: **Systém pro zpracování dat z chytrých zařízení**
Kategorie: Informační systémy
Akademický rok: 2023/24

Zadání:

1. Prostudujte oblast internetu věcí (*Internet of Things*, IoT) a možnosti využití klíčových ukazatelů výkonnosti (*Key performance indicator*, KPI) pro sledování chytrých zařízení.
2. Prostudujte problematiku výkonnosti informačních systémů a proveďte průzkum současných technologií pro vývoj výkonných informačních systémů (např. Rust, Go, apod.).
3. Proveďte analýzu požadavků na systém pro zpracování a vyhodnocování velkých objemů dat získaných ze senzorů chytrých zařízení.
4. Na základě požadavků z bodu 3 navrhnete architekturu systému, způsob uložení dat ze zařízení a vyhodnocení KPI.
5. Navržený systém implementujte.
6. Výsledný systém otestujte a vyhodnoťte jeho výkonnost.

Literatura:

- Greengard, S. (2015). *The Internet of Things*. MIT Press.
- Kirimtat, A., Krejcar, O., Kertesz, A., & Tasgetiren, M. F. (2020). Future trends and current state of smart city concepts: A survey. *IEEE access*, 8.
- Parmenter, D. (2015). *Key performance indicators: developing, implementing, and using winning KPIs*. John Wiley & Sons.

Při obhajobě semestrální části projektu je požadováno:
Body 1 - 4.

Podrobné závazné pokyny pro vypracování práce viz <https://www.fit.vut.cz/study/theses/>

Vedoucí práce: **Hynek Jiří, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1.11.2023
Termín pro odevzdání: 9.5.2024
Datum schválení: 30.10.2023

Abstrakt

Tato práce se zabývá návrhem, implementací a testováním nového informačního systému pro zpracování značných objemů dat z chytrých zařízení pomocí uživatelských kritérií – terminologicky klíčových indikátorů výkonnosti. Práce popisuje oblast internetu věcí a chytrých měst a pojednává o klíčových indikátorech výkonnosti a jejich použitelnosti pro sledování chytrých zařízení a transformaci dat z těchto zařízení na prakticky uplatnitelné poznatky. Dalším významným tématem práce jsou informační systémy, jejich architektura a výkonnost – zde práce popisuje různé pojmy a aspekty pomocí kterých lze definovat a vyhodnocovat výkonnost software, zejména informačních systémů a webových služeb. Následně se již práce zabývá samotným systémem a to nejprve z hlediska specifikace požadavků, určených rozбором potenciálních uživatelů a jejich potřeb a také rozбором již existujících řešení a jejich nedostatků. Významné požadavky na nový systém zahrnují proudové zpracování dat z chytrých zařízení v reálném čase, vysokou kapacitu systému, stabilitu systému a rozšiřitelnost. Posléze je popsán návrh systému s důrazem na datový model systému, proces vyhodnocení klíčových indikátorů výkonnosti a architekturu systému a poté implementace, s důrazem na implementaci klientské strany systému, implementaci jednotlivých služeb tvořících backend systému, komunikaci mezi těmito službami pomocí technologie RabbitMQ atd. Poslední část práce, věnovaná testování systému, se zaměřuje na výkonnost systému, resp. na propustnost jednotlivých služeb při zpracování zpráv z chytrých zařízení, optimalizaci propustnosti, přínos horizontálního škálování a celosystémovou latenci.

Abstract

This thesis deals with the design, implementation, and testing of a new information system for processing significant volumes of data from smart devices using user criteria – terminologically Key Performance Indicators. The thesis describes the domain of the Internet of Things and Smart Cities and discusses Key Performance Indicators and their applicability for monitoring smart devices and transforming data from these devices into practically applicable knowledge. Another significant topic of the thesis is information systems, their architecture, and performance – here the thesis describes various concepts and aspects through which it is possible to define and evaluate the performance of software, especially information systems and web services. Subsequently, the thesis deals with the system itself, first from the perspective of requirement specification, determined by analysis of potential users and their needs, and also analysis of existing solutions and their shortcomings. Significant requirements for the new system include stream processing of data from smart devices in real-time, high system capacity, system stability, and expandability. Later, the design of the system is described with an emphasis on the data model of the system, the process of evaluating Key Performance Indicators, and the architecture of the system, followed by implementation with an emphasis on the implementation of the client side of the system, the implementation of individual services forming the backend of the system, communication between these services using RabbitMQ technology, etc. The final part of the thesis, dedicated to testing the system, focuses on system performance, i.e., the throughput of individual services in processing messages from smart devices, optimization of throughput, benefits of horizontal scaling, and system-wide latency.

Klíčová slova

Internet věcí, Chytrá města, Klíčové indikátory výkonnosti, Informační systémy, Architektura informačních systémů, Výkonnost, Proudové zpracování dat, Zpracování dat v reálném čase, Propustnost, Škálovatelnost, Souběžnost, Paralelismus, MQTT, Go, React, TypeScript, GraphQL, RabbitMQ

Keywords

Internet of Things, Smart Cities, Key Performance Indicators, Information Systems, Architecture of Information Systems, Performance, Stream Processing of Data, Real-time Data Processing, Throughput, Scalability, Concurrency, Parallelism, MQTT, Go, React, TypeScript, GraphQL, RabbitMQ

Citace

BUREŠ, Michal. *Systém pro zpracování dat z chytrých zařízení*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Hynek, Ph.D.

System pro zpracování dat z chytrých zařízení

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Hynka, Ph.D. Další informace mi poskytl pan Ing. Petr John. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....

Michal Bureš

24. července 2024

Poděkování

Chtěl bych poděkovat vedoucímu práce – panu Ing. Jiřímu Hynkovi, Ph.D., za pomoc, rady a časté konzultace. Za odbornou pomoc, účast na konzultacích, a rady z hlediska praktických aspektů této práce bych chtěl poděkovat panu Ing. Petru Johnovi.

Obsah

1	Úvod	3
2	Internet věcí, chytrá města a klíčové ukazatele výkonnosti	4
2.1	Internet věcí	4
2.2	Chytrá města	9
2.3	Klíčové ukazatele výkonnosti	10
3	Informační systémy: architektura, výkonnost a technologie	12
3.1	Architektura informačních systémů	12
3.2	Výkonnost informačních systémů	15
3.3	Technologie	20
4	Analýza požadavků na systém	22
4.1	Rozbor existujících řešení	23
4.2	Požadavky na systém	25
4.3	Případy užití systému	27
5	Návrh systému	31
5.1	Model klíčových indikátorů výkonnosti	32
5.2	Vyhodnocení klíčových ukazatelů výkonnosti	35
5.3	Stabilita a propustnost systému	37
5.4	Trvalé úložiště dat – relační databáze	38
5.5	Architektura systému	40
6	Implementace systému	42
6.1	Klientská strana a uživatelské rozhraní	42
6.2	Aplikační rozhraní na bázi technologie GraphQL	46
6.3	Služby tvořící backend	47
7	Testování systému	57
7.1	Měření výkonnosti systému	58
7.2	Optimalizace propustnosti systému	62
7.3	Zvyšování propustnosti systému škálováním	63
7.4	Celosystémová latence	65
7.5	Zhodnocení	65
8	Závěr	66
	Literatura	68

Seznam obrázků

2.1	Vztahy mezi pojmy z oblasti IoT	5
2.2	Protokol MQTT a mechanismus Publish/Subscribe	7
2.3	Oblasti, do kterých zasahuje koncept chytrého města	9
3.1	Schéma tříúrovňové architektury	13
3.2	Monolitická aplikace a ekosystém mikroslužeb	14
3.3	Statistické charakteristiky doby odezvy	18
4.1	Ukázka vizualizace vyhodnocení klíčových identifikátorů výkonnosti	25
4.2	Diagram případů užití systému – 1. část	29
4.3	Diagram případů užití systému – 2. část	30
5.1	Zjednodušený diagram vztahů mezi datovými entitami	32
5.2	Příklad definice klíčového indikátoru výkonnosti	34
5.3	Proces vyhodnocení klíčových indikátorů výkonnosti	36
5.4	ER diagram – konceptuální model relační databáze	39
5.5	Diagram navržené architektury systému	41
6.1	Editor definic klíčových indikátorů výkonnosti	44
6.2	Obrazovka „SD instances“	45
6.3	Sekce zobrazující vyhodnocení klíčových indikátorů výkonnosti	45
6.4	Diagram komunikace mezi službami pomocí RabbitMQ.	49
7.1	Grafana – dashboard	58
7.2	Rychlost plnění fronty <code>kpi-fulfillment-check-requests</code>	60
7.3	Pokles množství zpráv ve frontě <code>kpi-fulfillment-check-requests</code>	61
7.4	Nárůst propustnosti segmentu zpracování dat s ohledem na počet instancí služby <code>Message processing unit</code> : proporcionální nárůst $\times$ realita	64

Kapitola 1

Úvod

Internet věcí transformuje průmysl, posouvá limity ostatních technologií a technologických konceptů a mění náš přístup k technologii i světu okolo nás. Z objektů fyzického světa tvoří datové body, což umožňuje rozsáhlejší studie a analýzy, na jejichž základě byly zaznamenány doposud neznámé úkazy a trendy. Je také základem chytrých měst a průmyslu 4.0 – domén s přímým dopadem na lidskou populaci. Nic z výše uvedeného by se ale neobešlo bez nasazení značného množství chytrých zařízení a zpracování narůstajícího objemu dat, která tato chytrá zařízení generují. Tato data je také třeba přeměnit na prakticky uplatnitelné poznatky. Žádoucí je i zjednodušení reprezentace komplexní infrastruktury, kterou nasazená chytrá zařízení tvoří. To zajišťuje rozličný software – od jednoduchých aplikací, přes informační systémy, až po komplexní platformy. Otázkou je, jestli dostatečně dobře, resp. jestli je reprezentace výsledků zpracování dat z chytrých zařízení srozumitelná, a jestli je zpracování dat do výsledné podoby dostatečně efektivní.

Primárním cílem práce bylo na tuto otázku nejen odpovědět, ale také na základě průzkumu relevantních oblastí a materiálů navrhnout nový, výkonný informační systém pro zpracování dat z chytrých zařízení, který nebude trpět běžnými nedostatky dnes dostupných řešení, a tento systém následně také implementovat a patřičně otestovat. Výsledný informační systém by měl také posloužit jako referenční bod pro tvůrce dalších aplikací, systémů a platform určených pro zpracování dat z chytrých zařízení. Toho má být dosaženo aplikací nových strategií a postupů, použitím moderních technologií, a důkladným zhodnocením jejich přínosu. Dalším z cílů práce bylo prozkoumat možnost sledování chytrých zařízení a přeměny dat z těchto zařízení na prakticky uplatnitelné poznatky pomocí metody známé především z byznysu – klíčových indikátorů výkonnosti.

V kapitole č. 2 je nejprve rozebrána problematika internetu věcí, chytrých měst a klíčových indikátorů výkonnosti, přičemž sekce týkající se klíčových indikátorů výkonnosti pojednává i o jejich použitelnosti pro sledování chytrých zařízení. Kapitola č. 3 se pak zaměřuje na architekturu informačních systémů, výkonnost informačních systémů a moderní technologie – zejména programovací jazyky, vhodné pro vývoj výkonných informačních systémů. Rozbor současných řešení a analýzu požadavků na nový, výkonný systém pro zpracování dat z chytrých zařízení lze nalézt v kapitole č. 4. Následující kapitola – kapitola č. 5, poté popisuje návrh tohoto systému. Implementaci systému, z hlediska aplikace konkrétních postupů a technologií, popisuje kapitola č. 6 a postup při testování systému se zaměřením na jeho výkonnost zase rozebírá kapitola č. 7. Obsahem poslední kapitoly – kapitoly č. 8, je závěr této práce.

Kapitola 2

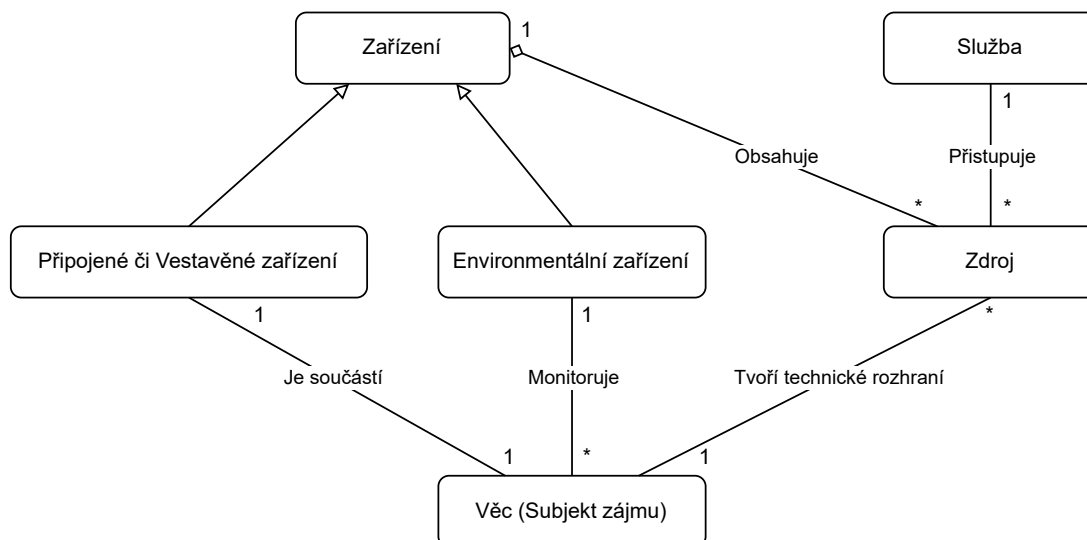
Internet věcí, chytrá města a klíčové ukazatele výkonnosti

2.1 Internet věcí

Podle S. Greengarda [12] lze *internet věcí*, *IoT* z angl. *Internet of Things*, popsat jako věci či objekty propojené v globálním rámci výpočtů, úložiště a komunikace. Následně přichází ještě s druhou definicí, podle které je internet věcí rozsáhlou sítí propojených věcí, které si navzájem předávají data a poskytují je databázím. Podle S. Hallera [16] se různé definice internetu věcí schází v jednom – podstatou internetu věcí je integrace našeho fyzického světa s virtuálním světem internetu. Existují fyzické objekty, které chceme sledovat či s nimi interagovat. Sem lze zahrnout nejen zboží, automobily, stroje či budovy, ale také zvířata a osoby. Všechny tyto živé i neživé prvky reálného světa a mnoho dalšího je možné chápat jako věci v rámci slovního spojení internet věcí. Přesněji se však podle S. Hallera [16] označují jako *subjekty zájmu*, z angl. *entities of interest*.

Pro sledování subjektů zájmu či interakci se subjekty zájmu je třeba je nejprve připojit k internetu. Podle S. Hallera [16] se tak děje pomocí *zařízení*, z angl. *devices*. Zařízení může být buďto součástí subjektu zájmu (připojené či vestavěné zařízení), čímž vznikne tzv. *chytrá věc*, z angl. *smart thing*, a nebo umístěno v prostředí, kde se jeden či vícero subjektů nachází (environmentální zařízení). Jako příklady zařízení lze uvést senzory, vestavěné počítače, mikročipy a chytré mobilní telefony. S. Greengard [12] popisuje dva druhy objektů připojených k internetu: *primárně fyzické* a *primárně digitální*. Objekty prvního typu, např. kniha, bota či pes, samy o sobě negenerují data a je třeba je doplnit o čip či senzor. Teprve pak se stanou součástí internetu věcí. Oproti tomu primárně digitální objekty, např. digitální fotografie či elektronická kniha, jsou součástí internetu věcí samy o sobě.

V momentě, kdy jsou subjekty zájmu připojeny k internetu, přichází na řadu *zdroje*, z angl. *resources*. Tak S. Haller [16] označuje výpočetní komponenty, které tvoří technické rozhraní subjektů zájmu. Tyto komponenty poskytují informace o subjektu, jako jsou identifikační údaje nebo snímaná data, a mohou rovněž nabízet funkce pro ovládání subjektu. Ke zdrojům přistupují přes zařízení *služby*, z angl. *services*, a přes tyto služby pak zbytek okolního světa. Kromě zpřístupnění zdrojů mohou služby nabízet i určitou úroveň agregace a abstrakce. Vztahy mezi výše zmíněnými pojmy z oblasti internetu věcí ilustruje UML diagram tříd v rámci obrázku 2.1.



Obrázek 2.1: UML diagram tříd ilustrující vztahy mezi pojmy z oblasti IoT.¹

2.1.1 Technologie, standardy a protokoly

S. Greengard [12] uvádí, že neexistuje žádný konkrétní standard, který by určil, jak bude internet věcí fungovat. Internet věcí podle něj stojí na mnoha standardech a technologiích, které spolu kooperují. Tento flexibilní základ napomáhá inovaci, ale také přináší vyšší úroveň complexity, problémy s nekompatibilitou a bezpečnostní rizika.

Jedním z klíčových aspektů internetu věcí je podle S. Greengarda [12] práce s primárně fyzickými objekty. Jde-li o jejich polohu, je třeba je označit. První technologií, kterou v tomto kontextu zmiňují S. Greengard, S. Haller i B. Stiller a spol. [12, 16, 38], je technologie *radio-frekvenční identifikace*, *RFID* z angl. *Radio-Frequency IDentification*. Podle S. Greengarda [12] je tato technologie založena na *štítcích*, z angl. *tags*, čipech a tištěných obvodech. Jakmile je objekt označen, lze jeho polohu určit pomocí RFID čtečky. Existují pasivní štítky bez baterií a aktivní štítky na baterie. Aktivní štítky jsou lépe sledovatelné, neboť pravidelně vysílají svou polohu. Zde se nejčastěji uplatňuje protokol DASH7, založený na standardu ISO/IEC 18000-7 [19], umožňující obousměrné přenosy dat na vzdálenost až 500 metrů. Funguje na frekvenci nižší než 1 GHz a nabízí nízkou latenci. Hodí se tak pro senzory, alarmy, průmyslové systémy atd. S. Greengard a B. Stiller a spol. [12, 38] dále zmiňují implantování RFID štítků do lidských těl, např. za účelem snazší autentizace osob. B. Stiller a spol. [38] pak zdůrazňují s tímto spojená bezpečnostní rizika. Na tato rizika reagují standardy ISO/IEC 18000-63 [21] a ISO/IEC 29167-10 [20], které obsahují mimo jiné mechanismy autentizace RFID štítků a čteček a kryptografické metody. Dalším standardem, který cílí na bezpečnost technologie RFID, je ISO/IEC 20248 [22]. Tento standard specifikuje použití digitálních podpisů.

S technologií RFID úzce souvisí technologie *NFC*, z angl. *Near Field Communication*, řídicí se standardem ISO/IEC 18092 [23]. Podle B. Stillera a spol. [38] je technologie NFC na technologii RFID přímo založená. NFC přináší přenos dat na relativně krátkou vzdálenost a umožňuje složitější interakce mezi zařízeními. Díky těmto vlastnostem je tato technologie

¹Diagram byl založen na diagramu z publikace S. Hallera [16].

vhodná zejména pro zabezpečené transakce a výměnu osobních údajů. To zahrnuje např. platby – k platbě dojde přiložením mobilního telefonu k platebnímu terminálu.

Dalšími komunikačními technologiemi, které S. Greengard a B. Stiller a spol. [12, 38] zmiňují, jsou *Wi-Fi*, *Bluetooth*, *Zigbee* a mobilní technologie (zejména *LTE-M* a *NB-IoT*). Ty zásadním způsobem rozšiřují možnosti komunikace IoT zařízení. Technologie *Wi-Fi*, která se řídí sadou standardů IEEE 802.11, se hodí zejména pro jednoduché propojení zařízení v rámci *lokální sítě*, *LAN* z angl. *Local Area Network*. Pracuje celosvětově v bezlicenčním frekvenčním pásmu 2.4 Ghz, z energetického hlediska je poměrně efektivní a nabízí vysokou rychlost přenosu dat v řádu stovek Megabitů za sekundu.

Technologie *Bluetooth* a *Zigbee* pak B. Stiller a spol. [38] uvádí s ohledem na komunikaci zařízení v rámci *osobní sítě*, *PAN* z angl. *Personal Area Network*. V oblasti internetu věcí se uplatňuje především technologie *Zigbee* a úspornější verze technologie *Bluetooth* známá jako *Bluetooth Low Energy*. V obou případech je hlavní motivací nízká energetická náročnost těchto technologií.

Přínos mobilních technologií pro internet věcí podle S. Greengarda [12] spočívá v tom, že poskytují robustní a flexibilní rámec pro komunikaci na delší vzdálenosti a ve větším měřítku. Zmiňuje se i o moderním standardu mobilních technologií – 5G. Ten podle něj přináší násobné zrychlení datových přenosů i zvýšení kapacity mobilních sítí co do počtu připojených zařízení. Značný význam S. Greengard [12] i B. Stiller a spol. [38] přisuzují také mobilním technologiím *NB-IoT* a *LTE-M*, navrženým specificky pro internet věcí. Jejich přednosti zahrnují již zmiňovanou nízkou energetickou náročnost, zejména ve srovnání se zbytkem mobilních technologií.

V rámci infrastruktury, kterou představují výše zmíněné komunikační technologie, operují aplikační protokoly. S. Greengard a B. Stiller a spol. [12, 38] zmiňují protokol *MQTT* (*Message Queuing Telemetry Transport*), protokol *CoAP* (*Constrained Application Protocol*), protokol *AMQP* (*Advanced Message Queuing Protocol*) a další. Vlastnosti protokolů pak detailněji rozebírá publikace N. Naika [31].

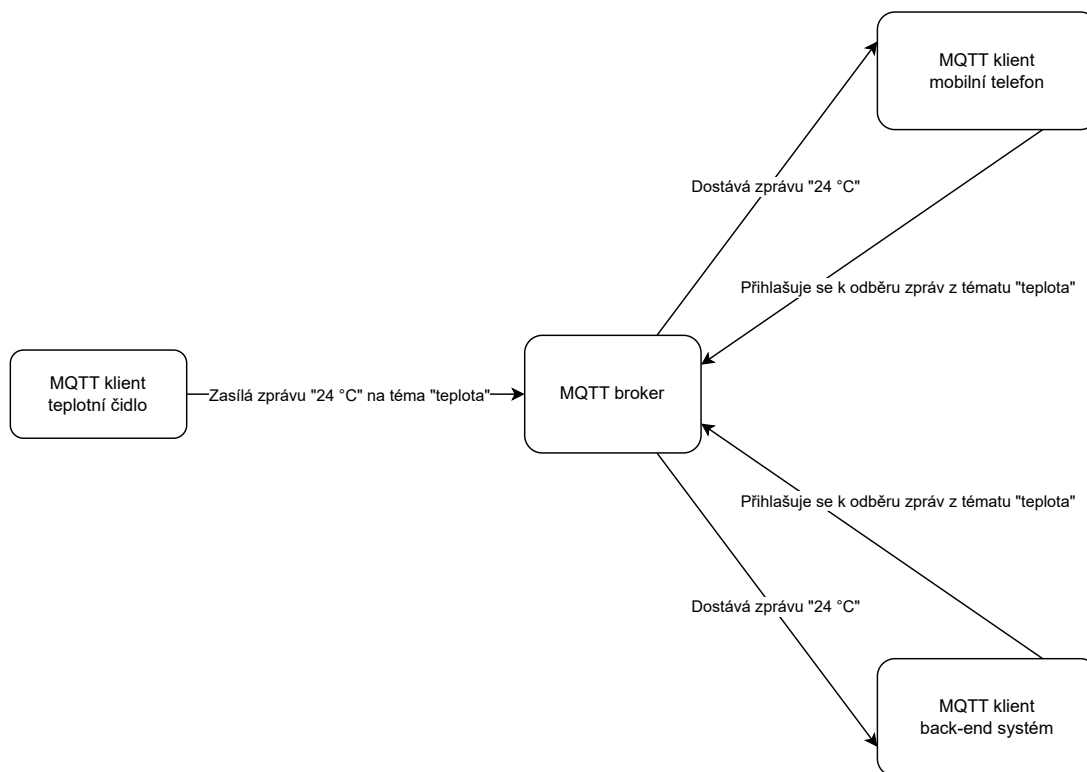
Protokol *MQTT* je binárním protokolem, který operuje nad transportním protokolem *TCP* (*Transmission Control Protocol*). Je určen pro odlehčenou komunikaci po sítích s omezenými zdroji a nabízí tři úrovně tzv. *kvality služeb*, *QoS* z angl. *Quality of Service*. Umožňuje tak kompromisy mezi spolehlivostí doručení zpráv a zatížením sítě. Tyto úrovně jsou:

- QoS 0: At most once – zpráva je doručena nanejvýš jedenkrát, tzn. zpráva může a nemusí být doručena.
- QoS 1: At least once – zpráva je doručena nejméně jedenkrát, tzn. doručení zprávy je zaručeno, ale může dojít k vícenásobnému doručení téže zprávy.
- QoS 2: Exactly once – zpráva je doručena právě jednou, tzn. doručení zprávy je zaručeno, aniž by docházelo k vícenásobnému doručení téže zprávy.

Kromě těchto úrovní kvality služeb minimalizuje protokol zátěž sítě i jinak, např. minimalistickými hlavičkami zpráv – hlavička zprávy zabírá pouhé 2 bajty, zatímco tělo zprávy může teoreticky pojmut až 256 Megabajtů. Režii protokolu naopak představuje otevírání a uzavírání spojení protokolu *TCP*.

Protokol *MQTT* funguje na bázi mechanismu *Publish/Subscribe*. Klient zašle zprávy centrálnímu serveru zvanému *broker* na danou adresu, resp. *téma*, z angl. *topic*, čímž se zpráva dostane k ostatním klientům přihlášeným na toto téma. Broker může zprávy na

tématech také uchovávat pro klienty, co se k tématům teprve přihlásí. Jako *Publish* se označuje situace, kdy klient zasílá zprávu na téma. Jako *Subscribe* pak situace, kdy se klient přihlašuje k odběru zpráv z daného tématu. Tento mechanismus spolu s konkrétním případem užití (informování zařízení přihlášených k patřičnému tématu MQTT brokeru o hodnotě naměřené teplotním čidlem) ilustruje diagram na obrázku 2.2.



Obrázek 2.2: Protokol MQTT a mechanismus Publish/Subscribe.²

Dalším protokolem, o kterém N. Naik [31] píše, je CoAP (Constrained Application Protocol). Protokol CoAP je založen na transportním protokolu *UDP* (*User Datagram Protocol*), jehož omezenou spolehlivost kompenzuje vlastním mechanismem opětovného zaslání zpráv. Místo tří úrovní kvality služeb, kterými disponuje MQTT, nabízí pouze dvě a to *confirmable messages* a *non-confirmable messages*. V prvním případě je od příjemce vyžadováno potvrzení přijetí zprávy, v druhém nikoliv. Hlavička zpráv tohoto protokolu zabírá 4 bajty, přičemž velikost těla závisí na konkrétní implementaci. Protokol podporuje dvojici mechanismů: *Resource/Observe*, což je varianta Publish/Subscribe, a *Request/Response*. Protokol CoAP je stejně jako protokol MQTT navržen pro odlehčenou komunikaci po sítích s omezenými zdroji. Protokol je také interoperabilní³ s aplikačním protokolem *HTTP* (*HyperText Transfer Protocol*) a webovými službami založenými na architektonickém stylu *REST* (*REpresentational State Transfer*).

Třetím důležitým aplikačním protokolem, který publikace N. Naika [31] zmiňuje, je protokol AMQP (Advanced Message Queuing Protocol). Ten byl navržen s ohledem na spolehlivost, bezpečnost a interoperabilitu. Využívá transportní protokol TCP, podporuje mechanismy Publish/Subscribe a Request/Response a stejně jako CoAP nabízí dvě úrovně

²Diagram je obdobou diagramu dostupného na stránce <https://mqtt.org/>.

³kompatibilní s, či schopen integrace s

kvality služeb: spolehlivější *Settle Format* a méně spolehlivý *Unsettle Format*. Oproti protokolům MQTT a CoAP je tento protokol obsáhlejší, robustnější a nabízí širší škálu funkcí. Hodí se tak především pro složitější scénáře komunikace a výkonnější sítě s více zdroji.

2.1.2 Přínos a aplikace

Podle S. Greengarda [12] nelze přínos internetu věcí podceňovat, neboť transformuje průmysl, posouvá limity ostatních technologií a technologických konceptů a mění náš přístup k technologii i světu okolo nás. Schopnost internetu věcí označovat fyzické objekty a transformovat je tak na datové body, lidstvu umožnila provádět podstatně rozsáhlejší studie a analýzy, na jejichž základě jsme zaznamenali určité úkazy a trendy, kterých bychom si jinak neměli šanci všimnout. Tento rozměr internetu věcí nastínil K. Ashton⁴ již v roce 2009. Ve svém článku [2] uváděl, že téměř veškerá data dostupná na internetu byla vytvořena lidmi: psaním na klávesnici, stisknutím tlačítka záznamu, pořízením digitálního snímku či naskenováním čárového kódu. Lidé jsou podle něj ovšem pomalí a nepřesní a nehodí se tak k zaznamenávání informací o světě. Místo toho bychom měli počítačům poskytnout prostředky, aby mohly shromažďovat informace o světě i bez naší pomoci. Tyto prostředky podle S. Greengarda [12] internet věcí úspěšně zajistil. Počítače a mnohá další zařízení tak dokážou pracovat bez nutnosti lidské obsluhy či dohledu, což vede k úsporám času, peněz a dalších zdrojů.

Podle F. Griffithse a spol. [14] je další doménou, kde má internet věcí klíčovou roli, *průmysl 4.0*. Ten definují jako koncept propojující výrobu s digitalizací. Mezi další podstatné aspekty průmyslu 4.0 patří rozšíření tzv. *propojených strojů*, z angl. *connected machines*, komunikace *stroj-stroj*, M2M z angl. *machine-to-machine*, augmentace⁵ strojů pomocí software, zabezpečení sítí a datová analýza. Role internetu věcí zde spočívá mimo jiné v postupné transformaci jednotlivých odvětví. Např. ohledně automobilního průmyslu, jenž je obecně považován za jedno z nejpokrokovějších průmyslových odvětví, se podle F. Griffithse a spol. [14] odhaduje, že přibližně 80% automobilů sestaví robotické systémy. Ty jsou v dnešní době často závislé na internetu věcí. F. Griffiths a spol. [14] také zmiňují novou generaci robotických systémů, označovanou jako tzv. *coboti*, z angl. *cobots*. Tyto systémy se vyznačují úzkou spoluprací s lidským pracovníkem a představují posun vstříc sofistikovanějším výrobním procesům. Není velkým překvapením, že coboti jsou do výroby zaváděni spolu s internetem věcí. Podle průzkumu IDC (International Data Corporation) uskutečněného v roce 2015, na který se F. Griffiths a [14] spol. odkazují, považuje internet věcí za záležitost strategického významu pro jejich podnikání 58% společností, přičemž 48% společností řešení na bázi internetu věcí již nasadilo. Konkrétní příklady těchto řešení nasazených reálnými společnostmi světového významu nabízí článek B. Buntze [6]. Třetím, avšak zdaleka ne posledním, konceptem využívajícím internet věcí jsou tzv. *chytrá města*, z angl. *smart cities*.

⁴K. Ashton je obecně považován za originálního autora slovního spojení Internet of Things, tedy internet věcí. Poprvé jej použil v roce 1999, když společnost Procter & Gamble prezentoval koncept zahrnující zvýšení efektivity dodavatelského řetězce pomocí technologie RFID.

⁵úprava či vylepšení

2.2 Chytrá města

Chytrá města jsou poměrně nový koncept, pro který existuje mnoho formálních definic. Např. podle C. Harrisona a spol. [18] je chytré město reakcí na potřebu propojit fyzickou infrastrukturu, sociální infrastrukturu, podnikatelskou infrastrukturu a infrastrukturu *informačních a komunikačních technologií, ICT* z angl. *Information and Communications Technology*, v městských oblastech. Podle S. N. Kondepudiho a spol. [28] je naplněním konceptu chytrého města moderní město, které těží z informačních a komunikačních technologií za účelem zvýšení kvality života jeho obyvatel. Klíčovou roli zde hraje kvalita služeb, které město svým obyvatelům poskytuje. Koncept chytrého města zasahuje do mnoha oblastí. Podle A. Kirimatové a spol. [26] jde především o dopravu, využití půdy resp. územní plánování, péči o životní prostředí, energetiku, hospodářství a správu a řízení města. M. Bauer a spol. [4] zmiňují dopravu, parkování, vodohospodářství, energetiku, odpady, osvětlení a bezpečnost. Koncept chytrého města spolu s oblastmi, do kterých zasahuje, také ilustruje infografika na obrázku 2.3.



Obrázek 2.3: Oblasti, do kterých zasahuje koncept chytrého města.⁶

Jako příklad úspěšné realizace konceptu chytrého města M. Bauer a spol. [4] uvádějí brazilské město Rio de Janeiro, které nasadilo rozsáhlou infrastrukturu internetu věcí za účelem zvýšení bezpečnosti, ochrany před přírodními katastrofami a zkvalitnění služeb poskytovaných obyvatelům. Tato infrastruktura zahrnuje veřejné přístupové body Wi-Fi, stanice NB-IoT, kamery CCTV, uzly CityIQ vybavené různými senzory, chytré osvětlení a různé druhy senzorů, např. senzory závlahy, GPS senzory atd. Dalším příkladem je pak čínské město Shenzen, které vyniká tím, jak kombinuje řešení na bázi internetu věcí s dalšími technologiemi. Těmi jsou např. tzv. *výpočty v cloudu*, z angl. *cloud computing* či *umělá inteligence*, z angl. *artificial intelligence*. Za zmínku stojí také sofistikovaný systém kamer a senzorů a platforma „společenského kreditu“.

M. Bauer a spol. [4] zmiňují také komerční platformy internetu věcí určené pro chytrá města. Principem těchto platforem je připojení městských IoT zařízení do poskytovatelem provozovaného cloudu. Senzory sem převážně zasílají data za účelem uložení, analýzy a vizualizace. Jiná zařízení odtud mohou data čerpat či přijímat příkazy.

⁶Infografika pochází ze stránky <https://czechsmartcitycluster.com>.

2.2.1 Sledování chytrých zařízení

Podle P. Belliniho a spol. [5] souvisí správa a sledování chytrých zařízení s potřebou komplexní agregace a analýzy dat. Ta je nezbytná pro informovaná rozhodnutí a poskytování efektivních služeb – klíčové aspekty chytrých měst. P. Bellini a spol. [5] zmiňují různé formáty dat, které je potřeba zpracovat a začlenit do jednotného systému chytrého města. Důležitá jsou zejména data ze senzorů, jejichž sledování umožňuje mimo jiné činit automatická rozhodnutí v reálném čase, která hrají klíčovou roli například v dopravě či pohotovostních službách.

Konkrétní příklad optimalizace reakčního času pohotovostních služeb chytrého města pomocí sledování chytrých zařízení a efektivní vizualizace založené na datech z těchto zařízení uvádí publikace A. Lavalleové a spol. [29]. Subjektem tohoto příkladu je hasičský sbor. Metoda navržená autory publikace kombinuje analýzu historických dat s analýzou dat z chytrých zařízení prováděnou v reálném čase. Výstupem této kombinované analýzy jsou efektivní vizualizace, na jejichž základě je možné zdokonalit správu zdrojů a reakční čas hasičského sboru.

Podle L. Pappasové a spol. [32] je třeba data transformovat na prakticky uplatnitelné poznatky. Klíčem má být korektní vizualizace, tedy veškerá relevantní data v podobě usnadňující jejich správné pochopení. Ideálním nástrojem pro seskupení podstatných dat je tzv. dashboard, inspirovaný palubními deskami aut a letadel. Užitečnost dashboardu pak určuje především jeho obsah. Tím mohou být např. tabulky, grafy či vizuální reprezentace klíčových indikátorů výkonnosti.

2.3 Klíčové ukazatele výkonnosti

Podle D. Parmentera [33] jsou klíčové ukazatele výkonnosti, KPIs z angl. *Key Performance Indicators*, jedním ze čtyř typů výkonnostních metrik uplatňovaných nejčastěji ve firmách a dalších organizacích za účelem optimalizace jejich chodu. Tyto čtyři typy se dále dělí na ukazatele výsledků a ukazatele výkonnosti.

Jako ukazatele výsledků se označují ty metriky, které managementu organizací pomáhají při pohledu na celkové výsledky, povětšinou dané spoluprací mnoha různých týmů. Neumožňují však přijmout konkrétní opatření – není totiž jasné, kdo či který tým je za celkové výsledky zodpovědný.

Druhou skupinu – ukazatele výkonnosti, tvoří metriky, které lze přiřadit ke konkrétnímu týmu či skupině týmů pracujících na společném cíli. Tyto metriky tak přinášejí přehlednost a jasně danou zodpovědnost.

Za klíčové ukazatele výkonu lze podle D. Parmentera [33] označit ty ukazatele výkonnosti, které jsou klíčové, tedy nejdůležitější pro současný i budoucí úspěch organizace. Klíčové ukazatele výkonnosti dále charakterizuje a od ostatních metrik odlišuje těchto sedm bodů:

1. Mají nefinanční charakter – měrnou jednotkou není měna.
2. Měří se často, tedy neustále, denně či nanejvýš týdně.
3. Zaměřují se na vrcholný management organizací, který na jejich základě jedná a přijímá opatření.
4. Jsou jednoduché – samotné metriky i související nápravná opatření by měl chápat veškerý personál organizace.

5. Za konkrétní metriku je zodpovědný konkrétní tým, který v případě potřeby realizuje nápravná opatření.
6. Metriky mají velký dopad na tzv. *kritické faktory úspěchu*, *CSFs* z angl. *Critical Success Factors*, a celkový úspěch organizace.
7. Metriky mají omezenou „temnou stranu“ – metriky by měly být řádně otestovány, aby se zajistilo, že mají pozitivní dopad, přičemž případné nezamýšlené důsledky jejich využití budou zanedbatelné.

2.3.1 Možnosti využití pro sledování chytrých zařízení

Publikace A. Lavallové a spol. [29] se zaměřuje především na efektivní vizualizaci dat z chytrých zařízení. Klíčové ukazatele výkonnosti jsou použity k měření míry dosažení cílů vizualizace dat, definovaných uživateli v rámci modelu uživatelských požadavků. Uživatelé definují nejprve požadavky a následně sadu klíčových ukazatelů výkonnosti, obsahující prahové hodnoty pro každý z cílů. Díky tomu je možné určit míru dosažení těchto cílů. Klíčové indikátory výkonu zde slouží jako podpůrný nástroj pro vyhodnocení úspěšnosti datové analýzy a vizualizačních technik s ohledem na prvořadě požadavky uživatelů. Propojují analýzu a vizualizaci dat z chytrých zařízení s vysokoúrovňovými uživatelskými požadavky.

Odlišný přístup prezentují M. Venuta a spol. [39]. Internet věcí a průmysl 4.0 mění podle nich mění povahu i použitelnost klíčových indikátorů výkonu. Uvádějí, že monitorování podniků typicky zajišťují informační systémy, které poskytují informace o několika klíčových ukazatelích výkonnosti. Klíčové ukazatele výkonnosti jsou nástrojem, který propojuje podniková data s obchodními cíli. Nasazení platform internetu věcí ovšem zahrnuje sběr obrovského množství dat, což umožňuje dynamickou vizualizaci informací a trendů a stejně tak i klíčových ukazatelů výkonnosti. M. Venuta a spol. [39] označují klíčové ukazatele výkonnosti jako o dynamické s ohledem na dynamičnost jejich výpočtu a vizualizace, založených na sběru dat v reálném čase. Tyto dynamické klíčové indikátory výkonu jsou nástrojem vhodným pro reprezentaci stavu strojů a zařízení a také pro identifikaci trendů v rámci datových proudů s rychle se měnícími hodnotami.

Nejjednodušší pohled na využití klíčových indikátorů výkonu nabízí publikace L. Pappasové a spol. [32]. Rolí klíčových indikátorů výkonu je ukázat, zda hodnoty obsažené v datech spadají do konkrétních, předem definovaných rozsahů, a jestli je vyžadována nějaká akce. Jde o nástroj umožňující snadnou a správnou interpretaci hodnot.

Na základě výše uvedených publikací lze konstatovat, že v kontextu sledování chytrých zařízení se klíčové ukazatele výkonnosti hodí jako nástroj pro reprezentaci stavu zařízení, snadnou interpretaci hodnot obsažených v datech, identifikaci trendů a propojení dat s cíli uživatelů či strategickými cíli organizací. Důležitá je i korektní vizuální reprezentace klíčových indikátorů výkonu, viz publikace L. Pappasové a spol. [32], která zmiňuje sémantické kódování pomocí barev.

Kapitola 3

Informační systémy: architektura, výkonnost a technologie

Podle R. K. Rainera a spol. [34] *informační systém*, *IS* z angl. *Information System*, shromažďuje, zpracovává, uchovává, analyzuje a šíří informace za určitým účelem. Smyslem informačního systému je dostat správné informace ke správným lidem ve správný čas. Zásadní je také množství informací a jejich formát. R. K. Rainer a spol. [34] zmiňují čtveřici komponent informačního systému. Těmito komponentami jsou:

- hardware – zařízení, která přijímají, zpracovávají a zobrazují data,
- software – program či kolekce programů umožňující práci s daty,
- databáze – kolekce souborů či tabulek obsahujících data,
- síť – propojovací systém umožňující výměnu informací a sdílení zdrojů mezi zařízeními.

Způsob organizace těchto komponent informačních systémů se pak typicky označuje pojmem *architektura*.

3.1 Architektura informačních systémů

Podle M. Fowlera [10] lze architekturu systémů chápat jako jejich členění na části na nejvyšší úrovni. Zásadní roli hraje také vzájemná interakce a komunikace těchto částí. Z praktického hlediska je pak architektura systému otázkou mnoha složitých rozhodnutí, která lze jen obtížně měnit. Důležitými pojmy jsou *vrstva*, z angl. *layer*, a *úroveň*¹, z angl. *tier*. Přestože jsou tak často používány, nejedná se o synonyma, tzn. jejich význam se liší. Pojem *vrstva* označuje logické rozdělení či organizaci zdrojového kódu. Pojem *úroveň* zase fyzické oddělení, tedy to, že software běží na různém hardware. Konkrétními příklady architektur podnikových aplikací a informačních systémů jsou *architektura klient-server*, *tříúrovňová architektura*, *třívrstvá architektura*, *architektura založená na mikroslužbách* a *serverless architektura*.

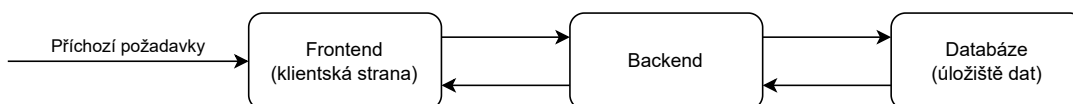
¹Angl. pojem *tier* se překládá také jako *fyzická vrstva*, přičemž angl. pojem *layer* se v tomto případě překládá jako *logická vrstva*.

3.1.1 Architektura klient-server

Podle M. Fowlera [10] jde o dvouúrovňovou architekturu. Jednou úrovní je klient – zařízení uživatele, a druhou server – zařízení provozovatele informačního systému. V tradičním pojetí této architektury jsou obě úrovně tvořeny jednou vrstvou. V případě klienta jde o tzv. *prezentační vrstvu*, zahrnující kód uživatelského rozhraní, a v případě serveru zase o tzv. *datovou vrstvu*, zahrnující nejčastěji relační databázi a s touto databází spojený kód v jazyce *SQL (Structured Query Language)*. M. Fowler [10] zmiňuje, že tato architektura byla široce populární, neboť byla vhodná pro dříve značně rozšířené systémy založené na zobrazování dat z databáze a jednoduchých modifikacích těchto dat. Závažný problém této architektury se objevil spolu s rozšířením tzv. *doménové* (popř. *byznys*) *logiky*, tedy složitějších operací nad daty, zahrnujících specifická pravidla, výpočty, validaci atd. Tuto logiku bylo těžké umístit jak na stranu klienta, tak na stranu serveru. V případě snahy o umístění doménové logiky na stranu klienta byl hlavním problémem špatně udržitelný kód navázaný na konkrétní obrazovky a další prvky uživatelského rozhraní. Na straně serveru byly hlavním limitujícím faktorem omezené možnosti jazyka SQL, resp. tzv. *uložených procedur* jazyka SQL.

3.1.2 Tříúrovňová architektura

Alternativu k dvouúrovňové architektuře klient-server představuje tříúrovňová architektura. Rozdělení do trojice úrovní, která tuto architekturu definuje, je podle S. J. Fowlerové [11] patrné téměř u všech dnešních systémů a aplikací. Jedná se o: *klientskou stranu* – *frontend*, *backend* a *databázi* – úložiště dat. Uživatel se systémem komunikuje pomocí klientské strany neboli frontendu. Backend na základě uživatelských požadavků, které obdržel od frontendu, z databáze načte patřičná data či data v databázi patřičným způsobem modifikuje a nakonec sestaví odpověď, kterou odešle na frontend. Ten na základě odpovědi z backendu informuje uživatele o výsledku požadované operace. Výše popsané úrovně tříúrovňové architektury a jejich interakci zachycuje schéma na obrázku 3.1.



Obrázek 3.1: Schéma tříúrovňové architektury.²

3.1.3 Třívrstvá architektura

Tříúrovňová architektura nabízí flexibilnější nasazení a provoz systému než dvouúrovňová architektura klient-server, ale není sama o sobě řešením problému umístění doménové logiky. Tím je až tzv. *třívrstvá* architektura, kterou lze chápat jako určitou nadstavbu tříúrovňové architektury, řešící organizaci zdrojového kódu a která se podle M. Fowlera [10] rychle rozšířila v době nástupu objektově-orientovaného paradigmatu programování³ a webového prohlížeče jako primárního cíle pro nasazení informačních systémů. Jednotlivé vrstvy této architektury se označují jako: *prezentační vrstva*, *doménová vrstva* a *datová vrstva*. Prezentační vrstva obsahuje uživatelské rozhraní systému a slouží tak pro obsluhu

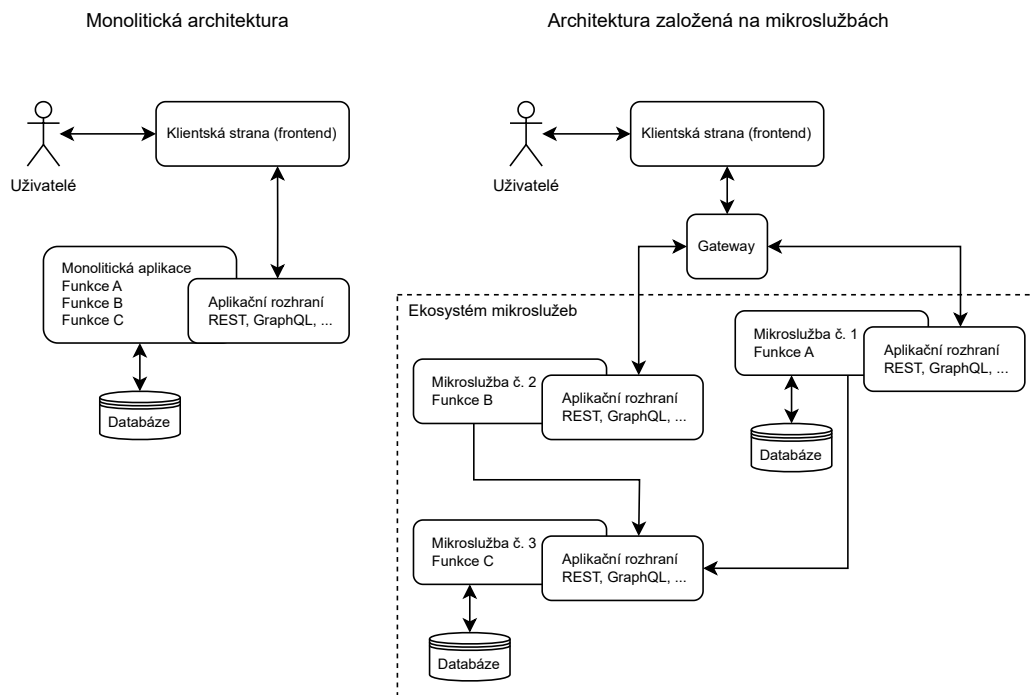
²Schéma je obdobou schématu z knihy S. J. Fowlerové [11].

³M. Fowler [10] v tomto kontextu zmiňuje především programovací jazyk Java, viz <https://www.java.com/>.

uživatelské interakce se systémem. To zahrnuje příjem uživatelských vstupů a zobrazení relevantních informací. Doménová vrstva, umístěná „uprostřed“, se zabývá doménovou logikou a s ní spojeným zpracováním dat. Třetí, datová vrstva slouží pro obsluhu trvalého úložiště dat a komunikaci s dalšími systémy a zdroji dat.

3.1.4 Architektura založená na mikroslužbách

Poměrně moderní architekturou je architektura založená na tzv. *mikroslužbách*, z angl. *microservices* – hlavní téma knihy S. J. Fowlerové [11]. S architekturou založenou na mikroslužbách úzce souvisí tzv. *monolitická architektura*, jejímž principem je sdružení veškeré funkcionality systému do jedné aplikace s jednou *základnou zdrojového kódu*, z angl. *codebase*, která je nasazována v jeden okamžik. Protipól tohoto přístupu představují mikroslužby – „malé“ aplikace, které jsou zodpovědné jen za určitou část funkcionality – ideálně zajišťují jen jednu jedinou funkci a tvoří je aplikační logika, aplikační rozhraní na bázi REST, GraphQL, gRPC atd., a často i oddělená databáze. S. J. Fowlerová [11] dále uvádí, že každá mikroslužba má být vyvíjena a nasazována nezávisle na ostatních mikroslužbách. Celkovou funkcionalitu systému má v rámci architektury založené na mikroslužbách zajišťovat tzv. *ekosystém mikroslužeb*, z angl. *microservice ecosystem*, který jednotlivé mikroslužby dohromady tvoří. Rozdíly mezi monolitickou aplikací a ekosystémem mikroslužeb znázorňuje schéma na obrázku 3.2.



Obrázek 3.2: Monolitická aplikace a ekosystém mikroslužeb.⁴

Za povšimnutí zde stojí především tzv. *gateway* – centrální aplikační rozhraní ekosystému mikroslužeb, se kterým komunikuje klientská strana systému a které zodpovídá za směrování požadavků mezi mikroslužbami, autentizaci atd. U monolitické aplikace nic jako *gateway* není, neboť klientská strana systému s monolitickou aplikací komunikuje napřímo.

⁴Schéma bylo inspirováno knihou S. J. Fowlerové [11] a článkem P. Karwatky [25].

Přechod z monolitické architektury na architekturu založenou na mikroslužbách podle S. J. Fowlerové [11] přináší mnohé benefity, např. lepší škálovatelnost systému, zvýšenou produktivitu vývojářských týmů a jednodušší nasazení systému. Protiváhu těchto benefitů pak představují především značné požadavky na infrastrukturu organizace, či latence spojená s komunikací po síti.

3.1.5 Serverless architektura

Jednou z nejmodernějších architektur je tzv. serverless architektura. Podle R. A. P. Rajana [35] se její podstatou rozložení systému či aplikace na *spouštěče*, z angl. *triggers*, a *akce*, z angl. *actions*. Předností této architektury je optimalizace využití výpočetních prostředků a s tím spojené snížení provozních nákladů. Systém či aplikace založená na serverless architektuře využívá zdroje pouze v rámci běhu akcí, tedy bezstavových funkcí, které jsou spouštěny když dojde k určité události – princip spouštěče. Další výhodou této architektury je pak to, že vše kromě spouštěčů a akcí je v kompetenci poskytovatele cloudových služeb. Příkladem prostředí určeného pro práci s touto architekturou je prostředí AWS Lambda⁵ spadající pod platformu Amazon Web Services od společnosti Amazon.

3.2 Výkonnost informačních systémů

Výkonnost informačních systémů je sofistikované téma, na které lze pohlížet mnoha způsoby. Např. skrze jejich roli v rámci firem a organizací. Tímto směrem se vydává publikace J. C.-J. Changa a spol. [7]. Přichází s teoretickým modelem, který zdůrazňuje úlohu informačních systémů při zvyšování efektivity podnikových procesů a výkonnosti organizace. Publikace tak nabízí komplexní a metodologicky přísný přístup a obchodně orientovaný pohled, který integruje technické aspekty informačních systémů do širšího kontextu organizací a jejich strategií.

Podle M. Fowlera [10] je třeba o výkonnosti tzv. *podnikových aplikací*, z angl. *enterprise applications*, a informačních systémů uvažovat v souvislosti s jejich architekturou. Výkonnostní dopady některých rozhodnutí týkajících se architektury může být totiž obtížné či dokonce zcela nemožné řešit pomocí pozdější optimalizace. Nebrání-li tomu architektura systému, je žádoucí zavést systematický proces optimalizace na základě skutečných, spolehlivě naměřených údajů o výkonnosti. M. Fowler [10] také zmiňuje význam konfigurace systému – výrazné změny konfigurace systému mohou anulovat přínos předchozích optimalizací. V návaznosti na změny konfigurace je tedy potřeba výkonnost systému opětovně vyhodnotit a provést potřebné úpravy.

M. Fowler a M. Kleppmann [10, 27] uvádí určité pojmy, které s výkonností úzce souvisí, nebo jsou k ní komplementární. Pochopení těchto pojmů umožňuje pochopení výkonnosti systémů, mimo jiné informačních, v širších souvislostech. Těmito pojmy jsou:

- *Doba odezvy*, z angl. *response time*, je podle M. Fowlera [10] čas potřebný pro zpracování požadavku zvenčí. Tím může být akce uživatelského rozhraní či požadavek odeslaný na *aplikační programovací rozhraní*, *API* z angl. *Application Programming Interface*, serveru. Podle M. Kleppmanna [27] doba odezvy systému dána tím, jak dlouho zabere zpracování požadavku. Tedy dobou, po kterou požadavek čeká na zpracování a zpožděním na síti.

⁵Viz <https://aws.amazon.com/lambda/>.

- *Responzivita* je podle M. Fowlera [10] otázkou času, který uplyne, než systém poskytne zpětnou vazbu ohledně přijatého požadavku. Touto zpětnou vazbou může být vizuální prvek uživatelského rozhraní, např. tzv. *ukazatel průběhu*, z angl. *progress bar*. Podle M. Fowlera [10] má vysoká responzivita systému, tedy krátká doba, než systém zohlední nový požadavek, pozitivní vliv na *uživatelský prožitek*, *UX* z angl. *User eXperience*. I. Grigorik [15] zase tvrdí, že systém by měl vizuální zpětnou vazbu poskytnout do 250 milisekund od ukončení dané uživatelské interakce.
- *Latence*, je podle M. Kleppmanna [27] zpoždění, přesněji zpoždění na síti, a jde podle něj o zásadní faktor při návrhu datově intenzivních systémů. I. Grigorik [15] zase uvádí, že latence je kritickým faktorem limitujícím výkonnost webových služeb a aplikací. Pro zlepšení uživatelského prožitku je podle něj potřeba zohlednit latenci a optimalizovat síťovou komunikaci na transportní i aplikační vrstvě *ISO/OSI modelu*. Tato doporučení podle něj platí zejména pro bezdrátové sítě, které se vyznačují vysokou latencí a omezenou *šířkou přenosového pásma*, z angl. *bandwidth*.
- *Zátěž*, z angl. *load*, může mít podle M. Kleppmanna [27] různou podobu. Nejčastěji jde o množství uživatelů současně pracujících se systémem, či objem dat, která musí systém zpracovat. M. Fowler [10] jako příklad zátěže uvádí počet uživatelů připojených k systému. Platí, že větší množství uživatelů připojených k systému představuje vyšší míru zátěže systému, stejně jako větší objem dat, která je třeba zpracovat. Podstatné je i to, že míra zátěže systému typicky není konstantní.
- *Kapacita*, z angl. *capacity*, je podle M. Fowlera [10] přímo navázána na zátěž. Určuje maximální míru zátěže, kterou systém zvládne, aniž by responzivita systému klesla pod přijatelnou úroveň.
- *Propustnost*, z angl. *throughput*, M. Fowler [10] definuje jako množství provedených úkonů za určitý čas. U podnikových aplikací a informačních systémů se nejčastěji uvádí v transakcích za sekundu – tps. U systémů, jejichž primárním účelem je zpracování dat či práce se soubory, ji lze uvádět v bajtech za sekundu – Bps, či jiných jednotkách. Důležitá je zejména závislost na čase.
- *Škálovatelnost*, z angl. *scalability*, je dle M. Kleppmanna [27] schopnost systému poradit si se zvýšenou mírou zátěže. Podle M. Fowlera [10] je to míra schopnosti systému využít dodatečné zdroje ke zvýšení výkonu či propustnosti. Jako zdroje M. Fowler označuje:
 - procesor (CPU),
 - operační paměť (RAM),
 - kapacitu a rychlost úložiště dat,
 - šířku pásma síťového připojení

Podle M. Kleppmanna, M. Fowlera i S. J. Fowlerové [27, 10, 11] lze systémy škálovat dvěma způsoby:

- *Vertikální škálování* – systém získá více zdrojů tím, že bude provozován na výkonnějším hardware, např. výkonnějším serveru, který disponuje procesorem s více jádry. Tento způsob škálování je jednodušší, ale má i určité nevýhody.

Jednou z výrazných nevýhod tohoto způsobu škálování je to, že pořizovací cena dvakrát výkonnějšího hardware, např. již zmíněného serveru, bude téměř jistě vyšší než dvojnásobek ceny hardware s polovičním výkonem.

- *Horizontální škálování* – systém získá více zdrojů tím, že bude provozován v rámci distribuovaného hardwarového systému, přičemž zátěž bude co možná nejrovnoměrněji rozdělována mezi jednotlivé stroje (nazývané *uzly*, z angl. *nodes*). Komponenta zajišťující rozdělení zátěže se nazývá *load balancer*. Složitost horizontálního škálování systémů závisí na jejich architektuře. Tento způsob škálování usnadňuje již zmíněná architektura založená na mikroslužbách, viz 3.1.4.

M. Kleppmann [27] v tomto kontextu uvádí dvojici otázek:

- Co se stane s výkonem systému, pokud bude mít systém k dispozici stále ty samé zdroje a zátěž se zvýší?
- Má-li výkon systému zůstat stejný i při zvýšené zátěži, o kolik musíme navýšit zdroje, které systém využívá?

Tyto otázky se ovšem netýkají škálovatelnosti. Místo toho směřují k poslednímu zásadnímu pojmu, kterým je *efektivita*, z angl. *efficiency*.

- Efektivita je podle M. Fowlera [10] vztahem mezi výkonem a množstvím zdrojů, které systém spotřebuje. Budou-li mít dva systémy k dispozici stejné zdroje, bude vždy výkonnější ten efektivnější. Efektivnější systém bude také lépe reagovat na vertikální škálování.

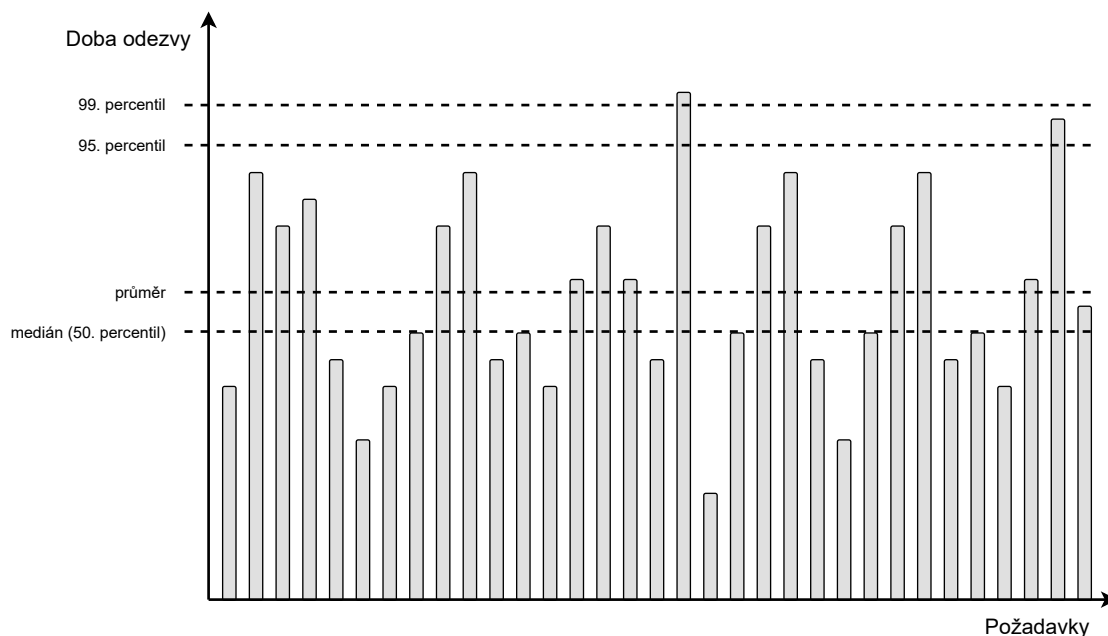
3.2.1 Výkonnostní metriky

Prerekvizitou pro hodnocení výkonnosti informačních systémů je ustanovení adekvátních metrik. Podle M. Kleppmanna [27] by měla metrika určená pro hodnocení výkonnosti systému odrážet jeho povahu. Jedná-li se o systém, jehož hlavním smyslem je zpracování dat⁶, je vhodné metriky odvozovat od propustnosti systému. Např.:

- počet záznamů zpracovaných za sekundu,
- čas potřebný pro zpracování souboru dat o určité, předem dané velikosti.

Jde-li o systém zaměřený na interakci s uživateli, je podle M. Kleppmanna [27] vhodnější výkonnostní metriku modelovat na bázi doby odezvy systému. Při transformaci doby odezvy systému na vhodnou metriku je potřeba dobu odezvy systému chápat nikoliv jako jednu hodnotu, ale rozdělení naměřených hodnot. V tento moment lze zavést vlastní metriku jako *kvantil* – charakteristiku datového souboru udávající hodnotu, kterou stanovená část hodnot nepřesahuje. Tedy např. 99. percentil je charakteristikou, určující hodnotu, kterou nepřesahuje 99 % hodnot v datovém souboru. Tyto charakteristiky ukazuje také graf na obrázku 3.3.

⁶Jako příklad uvádí Apache Hadoop, viz <https://hadoop.apache.org/>.



Obrázek 3.3: Statistické charakteristiky doby odezvy.⁷

S. J. Fowlerová [11] zmiňuje metriky pro hodnocení efektivity a výkonnosti mikroslužeb. První skupinou těchto metrik jsou metriky, které se vážou na infrastrukturu, v rámci které jsou mikroslužby provozovány. Patří sem využití systémových prostředků a zdrojů, jako jsou:

- procesor,
- operační paměť,
- *vlákna*, z angl. *threads*,
- *popisovače souborů*, z angl. *file descriptors*,
- *instance připojení k databázi*, z angl. *database connections*.

Tyto metriky by podle S. J. Fowlerové [11] měly doplňovat metriky navržené specificky pro danou mikroslužbu. Ty často obsahuje smlouva *SLA*, z angl. *Service Level Agreement*, a běžně se vážou na dobu odezvy a latenci.

V. Aranujo a spol. [1], kteří se zabývali hodnocením výkonnosti známé cloudové platformy pro aplikace internetu věcí FIWARE⁸, použili metriky založené na propustnosti, využití zdrojů a době odezvy. Přesněji šlo o:

- propustnost platformy při zpracování dat a ukládání těchto dat do databáze MongoDB,
- využití systémových prostředků jednotlivými komponentami platformy,
- dobu odezvy v kontextu dotazů směřovaných na komponentu platformy označovanou jako Orion Context Broker.

⁷Graf je obdobou grafu z knihy M. Kleppmanna [27].

⁸Viz <https://www.fiware.org/>.

3.2.2 Měření výkonnosti – benchmarking, profiling a monitoring

Výše popsané metriky je třeba zařadit do systematického procesu měření výkonnosti systému a následné optimalizace, který zmiňuje M. Fowler [10]. Tento proces by měl probíhat opakovaně a to buďto periodicky a nebo alespoň po každé výraznější změně implementace či konfigurace systému. Samotného měření se týká trojice pojmů, resp. technik: *benchmarking*, *profiling* a *monitoring*. Podle B. Gregga [13] je lze definovat následujícím způsobem.:

- Benchmarking – měření výkonnosti kontrolovaným způsobem, umožňující snadné srovnání výkonnostních dopadů dílčích rozhodnutí a včasnou identifikaci výkonnostních limitů – tedy předtím, než dojde k nasazení systému. Benchmarking lze rozdělit na *Micro-benchmarking* zaměřený na specifické komponenty systému a *Macro-benchmarking* zaměřený na celý systém.
- Profiling – měření výkonnosti systému pomocí nástrojů, které periodicky zaznamenávají určitou podmnožinu dostupných informací o cíli profilingu, přičemž nejčastějším cílem je procesor zařízení, na kterém systém běží. Výstupem profilingu je povědomí o náročnosti konkrétních segmentů a cest zdrojového kódu systému z hlediska využití procesoru a dalších systémových zdrojů. Toto povědomí lze ještě rozšířit doplněním samotného profilingu o vhodný vizualizační nástroj, např. tzv. *flame graph*.
- Monitoring – měření hodnot výkonnostních metrik po delší čas, které umožňuje porovnat momentálně naměřené hodnoty s hodnotami naměřenými v minulosti a také získat povědomí o reálné zátěži systému. Zatímco benchmarking a profiling jsou nejčastěji součástí ladění systému před nasazením, monitoring se typicky týká již nasazeného systému.

K měření výkonnosti systémů a následné vizualizaci naměřených hodnot lze použít různé nástroje a platformy, např.:

- Platforma společnosti Datadog – jedná se o populární, komerční platformu zaměřenou na monitoring webových služeb a infrastruktury.
- Zabbix – multiplatformní systém určený pro monitoring síťových prvků, cloudových řešení, aplikačních rozhraní, služeb, chytrých zařízení atd.
- Intel® VTune™ Profiler – pokročilý profiler určený pro počítače s instrukční sadou x86. Umožňuje identifikaci chyb vícevláknového programování a nedostatků z hlediska využití procesoru a operační paměti.
- Apache JMeter™ – open source nástroj zaměřený na benchmarking, resp. kontrolované, zátěžové testování webových aplikací.
- Grafana – open source nástroj zaměřený na vizualizaci dat z různých zdrojů, který je často používán v kombinaci s dalšími nástroji pro měření výkonnosti a zajišťuje vizualizaci naměřených hodnot. Populární kombinací představuje Grafana a open-source systém Prometheus se zaměřením na monitoring, který disponuje dimenzionálním datovým modelem, flexibilním dotazovacím jazykem *PromQL*, databází časových řad, z angl. *time-series database* atd.

3.3 Technologie

Kromě architektury jde u výkonných informačních systémů i o použité technologie. Značný dopad na výslednou výkonnost může mít i volba programovacího jazyka. Mezi programovací jazyky preferované při tvorbě vysoce výkonného software patří především C++ a také moderní jazyk Rust⁹. Podle bakalářské práce P. Karlssona [24] je společným rysem obou jazyků vysoký stupeň kontroly nad systémovými prostředky včetně manuální správy paměti. Díky manuální správě paměti se těchto jazyků netýká výkonnostní režie spojená s procesem zvaným *sběr odpadu*, z angl. *garbage collection*, na který mnohé jazyky v otázce správy paměti spoléhají.

Výhodou jazyka Rust oproti jazyku C++ je to, že dokáže lépe zajistit bezpečnost práce s pamětí. Podle A. Balasubramaniana a spol. [3] je to dáno robustním modelem tzv. *vlastnictví*, z angl. *ownership*, který je součástí jazyka Rust. V rámci tohoto modelu musí mít každá hodnota konkrétního „vlastníka“ či „majitele“. V momentě, kdy se vlastník hodnoty dostane mimo tzv. *scope*, tzn. stane se nedostupným, Rust automaticky uvolní paměť, ve které byla hodnota uložena. Tento koncept podle článku [8] dále rozšiřuje mechanismus tzv. *výpůjček*, z angl. *borrowing*. Tyto výpůjčky umožňují bezpečné sdílení dat bez změny vlastníka hodnoty. Jsou realizovány pomocí jednoho *měnitelného odkazu*, z angl. *mutable reference*, či vícero *neměnných odkazů*, z angl. *immutable reference*, a připomínají tak do jisté míry tzv. *chytré ukazatele*, z angl. *smart pointers*, v jazyce C++. Z omezení daných modelem vlastnictví jazyka Rust se pak může programátor „na vlastní nebezpečí“ vyvázat použitím klíčového slova `unsafe`.

Z hlediska výkonu jazyky srovnává již zmiňovaná bakalářská práce P. Karlssona [24]. Výkon jazyků měří v rámci tří úloh. Jedná se o:

- násobení matic,
- řazení čísel pomocí algoritmu merge sort,
- práci se soubory – zápis a čtení dat.

Jazyk C++ předčil jazyk Rust v násobení matic a čtení souborů, zatímco jazyk Rust se ukázal jako o něco lepší volba při řazení pomocí algoritmu merge sort a zejména při zápisu velkých objemů dat do souborů. Autor práce zároveň připomíná, že volbě programovacího jazyka by mělo předcházet zvážení konkrétního případu užití a všech relevantních okolností.

Dalším faktorem, který je žádoucí při volbě programovacího jazyka pro tvorbu výkonného software zohlednit, je úroveň podpory *vícevláknového programování*, z angl. *multithreaded programming*, kterou daný jazyk poskytuje. V rámci modelu vícevláknového programování lze podle M. Schilla a spol. [37] mluvit o dvou stěžejních konceptech: *souběžnosti*, z angl. *concurrency*, a *paralelismu*, z angl. *parallelism*. Přestože jsou tyto koncepty často zaměňovány a do určité míry se překrývají, jedná se skutečně o odlišné koncepty a každý z nich má své přednosti. V případě souběžnosti jde zejména o spolehlivou mezivláknovou komunikaci a integritu sdílených dat. V případě paralelismu pak o rozdělení výpočetně náročných úloh mezi vícero jader procesoru, což zvyšuje rychlost jejich zpracování.

Moderním programovacím jazykem, který vyniká v podpoře souběžnosti i paralelismu je jazyk Go¹⁰. Podle K. Cox-Budayové [9] je podpora souběžnosti a paralelismu v jazyce Go

⁹Viz <https://www.rust-lang.org/>.

¹⁰Viz <https://go.dev/>.

založena na tzv. *goroutines* a *kanálech*, z angl. *channels*. Tím se jazyk Go liší od mnoha dalších programovacích jazyků, které pro souběžnost používají vlákna operačního systému. Model jazyka Go má následující přednosti:

- Podpora souběžnosti na úrovni jazyka – podpora souběžnosti je integrována přímo do jazyka. Tzv. *primitiva* souběžnosti odpovídají struktuře jazyka i typickým scénářům použití jazyka. To vede k intuitivnějšímu programování, přičemž výsledné programy jsou méně náchylné na chyby.
- Odlehčená alternativa k vláknům – ve srovnání s vlákny operačního systému představují *goroutines* menší zátěž z hlediska systémových prostředků. Díky tomu je možné vytvářet a spravovat větší množství *goroutines*, než kolik by šlo vytvořit a spravovat vláken.
- Zjednodušená správa souběžnosti – kanály, které slouží pro komunikaci a synchronizaci mezi *goroutines*, spolu s příkazem `select` výrazně zjednodušují tradičně složitou správu souběžnosti.
- Seskupování kanálů – seskupování kanálů usnadňuje koordinaci vstupů a výstupů jednotlivých subsystémů. Tato úloha je v rámci modelů založených na vláknech podstatně složitější.

K. Cox-Budayová [9] uvádí, že podporu paralelismu zajišťuje běhové prostředí jazyka Go. Jednotlivé *goroutines* mapuje na hardwarové prostředky stroje a zohledňuje přitom další parametry, např. momentální zatížení stroje ostatními programy. Výsledkem je automatická optimalizace využití systémových zdrojů.

3.3.1 Další technologie a faktory

Je zřejmé, že kromě programovacího jazyka je zde mnoho dalších technologií, které mohou zásadním způsobem ovlivnit výkonnost výsledného software, potažmo informačního systému. Mezi tyto další technologie patří např. technologie použité pro implementaci aplikačního rozhraní systému – nejčastěji jde o REST¹¹, GraphQL, či gRPC, a také technologické rámce na kterých je založena databáze – SQL, *NoSQL*, či *NewSQL*. Značnou, i když méně patrnou, roli hrají i síťové protokoly, a to jak aplikační – nejčastěji HTTP, tak transportní – nejčastěji TCP či UDP, což detailně popisuje I. Grigorik [15]. Opomenout nelze ani synergii jednotlivých technologií a jejich správné použití programátorem – nedokonalá implementace systému často znehodnotí přínos jinak dobře zvolených technologií, které jsou při implementaci systému použity.

¹¹REST (REpresentational State Transfer) je spíše architektonický styl či paradigma, ale byl zde zmíněn pro úplnost.

Kapitola 4

Analýza požadavků na systém

Typické uživatele systému pro zpracování dat z chytrých zařízení si lze představit jako operátory, administrátory a další personál obsluhující konkrétní řešení z oblasti chytrých měst, průmyslu 4.0 či jiné oblasti, kde se počítá s nasazením a následným monitoringem značného počtu chytrých zařízení. S jejich standardními úkony, zahrnujícími monitoring zařízení, detekci a reporting chyb a problémů, údržbu a optimalizaci daného řešení atd., jim pomáhá rozličný software – aplikace, systémy, a komplexní platformy. O náležitostech tohoto software, konkrétně pak softwarových platform pro správu a řízení chytrých měst, pojednává publikace E. F. Z. Santany a spol. [36], přičemž jako obecné požadavky na tyto platformy publikace uvádí následující:

- Škálovatelnost – platforma by měla být připravená na narůstající objem dat ze zařízení, narůstající počet uživatelů atd.
- Správa dat – platforma by měla umožnit sběr, zpracování, ukládání, analýzu a vizualizaci dat ze zařízení.
- *Interoperabilita* – platformu by mělo být možné integrovat s dalšími aplikacemi a systémy a měla by být schopná pracovat s různými formáty dat.
- *Rozšiřitelnost* – platformu by mělo být možné snadno rozšířit o další funkce.

Pro lidské uživatele těchto platform, i další systémy a aplikace, se kterými jsou tyto platformy propojeny, je zásadní zpracování dat z chytrých zařízení do smysluplné podoby. To může podle publikace [36] probíhat dvěma způsoby:

- *Dávkové zpracování dat*, z angl. *batch processing* – data z chytrých zařízení jsou po nějaký čas shromažďována a následně v určitý moment hromadně zpracována.
- *Proudové zpracování dat*, z angl. *stream processing* – ke zpracování dat z chytrých zařízení dochází průběžně, přičemž každá položka dat je zpracována jakmile „dorazí“, resp. jakmile je k dispozici.

Je nesporné, že pro oba způsoby zpracování dat existují ideální scénáře. Důležitou roli hraje míra zpoždění, kterou lze tolerovat na straně jedné, a nákladnost provozu platformy na straně druhé. Platformy a systémy, které zpracování dat z chytrých zařízení provádí proudově, minimalizují zpoždění, tedy prodlevu od chvíle kdy jsou nová data dostupná po jejich zohlednění, ale jejich provoz může být nákladnější. U platform a systémů

provádějících dávkové zpracování dat je situace opačná. Podle publikace [36] u platforem pro správu a řízení chytrých měst převažuje dávkové zpracování dat z chytrých zařízení, přičemž absenci nástrojů umožňujících proudové zpracování dat autoři publikace označují za častý nedostatek, který může zapříčinit, že úkony a rozhodnutí personálu spravujícího chytré město se mohou opírat o zastaralá data. Benefity proudového zpracování dat z chytrých zařízení jsou pak ještě patrnější u platforem a systému sloužících personálu z dalších oblastí, jako je např. průmysl 4.0, a také platforem a systémů, které jsou propojeny s dalšími systémy a aplikacemi. Komunikaci mezi jednotlivými systémy ani různé automatické úkony totiž nezpomaluje lidský faktor a z okamžité dostupnosti výsledků zpracování dat z chytrých zařízení tak lze vytěžit maximum. U nových platforem a systémů pro zpracování dat z chytrých zařízení se tak proudové zpracování dat jeví jako další jasný požadavek.

Mezi obecnými požadavky na systém pro zpracování dat z chytrých zařízení nelze opomenout ani uživatelskou přívětivost systému. Podstatná je především vhodná vizualizace aktuálního stavu jednotlivých zařízení a přiměřená úroveň abstrakce, odpovídající rozsahu spravované infrastruktury, kterou chytrá zařízení tvoří. Patrné je i to, že uživatelé budou chtít úroveň abstrakce a charakter vizualizace měnit v závislosti na svých aktuálních potřebách a aktuálně prováděných úkonech. Chybět by neměly ani mechanismy pro agregaci výsledků zpracování dat, které zajistí, že systém zůstane přehledný bez ohledu na množství sledovaných zařízení, resp. objem dat z těchto zařízení.

4.1 Rozbor existujících řešení

Obecné požadavky systém pro zpracování dat z chytrých zařízení získané skrze rozbor potenciálních uživatelů systému a jejich potřeb a studium publikace E. F. Z. Santany a spol. [36] bylo žádoucí dále precizovat rozбором již existujících řešení s ohledem na to, v čem tato řešení vynikají a kde jsou naopak jejich „slabá místa“. Zde jsem se zaměřil na platformy pro správu a řízení chytrých měst, neboť na tuto oblast jsem se doposud zaměřoval nejvíce a mohl jsem zde při rozboru nejlépe aplikovat poznatky z již zmiňované publikace [36]. Konkrétně jsem se zaměřil na platformu FIWARE a platformu ACADA (Asset Control and Data Acquisition) od české společnosti Logimic¹. Platformu FIWARE jsem zvolil protože je populární, široce používaná a mohl jsem při jejím rozboru využít již zmiňovanou publikaci V. Aranuja a spol. [1]. Druhou platformu, tedy platformu ACADA, jsem zvolil, protože jsem měl v úmyslu do rozboru zahrnout i platformu od české společnosti. Společnost Logimic, která za platformou stojí, má navíc pro tuto bakalářskou práci speciální význam – nelze ji sice považovat za přímého zadavatele, ale výsledky práce by pro ni mohly být užitečné.

4.1.1 FIWARE

Jednou z předních softwarových platforem pro chytrá města, kterou podle oficiálních webových stránek využívají významná evropská města jako Vídeň či Brusel, je platforma FIWARE². Jednou z jejích předností je značná flexibilita ohledně reprezentace a vizualizace výsledků zpracování dat. Platforma kromě klasických nástrojů, jako jsou tabulky, grafy a dashboardy, nabízí také prostorovou vizualizaci, vizualizaci časových řad a především podporu pro integraci dalších nástrojů a knihoven³. Podle V. Aranuja a spol. [1] je tato

¹Viz <https://www.logimic.com/cs/>.

²Viz <https://www.fiware.org/>.

³Např. populárních knihoven jazyka JavaScript – D3.js (<https://d3js.org/>) a Three.js (<https://threejs.org/>).

platforma založena na modulární architektuře – skládá se z komponent označovaných jako *obecné aktivátory*, *GEs* z angl. *General Enablers*. Každá z těchto komponent zodpovídá za konkrétní oblast funkcionality, např. zabezpečení, správu zařízení, či správu dat. Klíčovou komponentu představuje tzv. *Orion Context Broker*, umožňující správu a zpracování dat z různých zdrojů. Při zpracování dat z různých zdrojů se uplatňují také další komponenty označované jako *IoT agenti*, z angl. *IoT agents*, které zprostředkovávají komunikaci mezi zařízeními a platformou. Právě tyto komponenty představují podle publikace [1] slabé místo z pohledu výkonnosti a škálovatelnosti platformy. Při zvýšené zátěži vykazovaly omezenou stabilitu a selhávaly. Problematické bylo též vertikální škálování, kdy dodání dalších zdrojů nevedlo k proporcionálnímu zvýšení výkonu. Podle V. Aranuja a spol. [1] byla výkonnost IoT agentů ovlivněná také technologií, která byla použita při jejich implementaci, tedy Node.js, běhový prostředí pro JavaScript, umožňujícím použití tohoto programovacího jazyka mimo webové prohlížeče. Prostředí Node.js totiž standardně operuje na jednom vlákně a do roku 2019⁴ nebylo snadné v rámci tohoto prostředí dosáhnout plnohodnotného využití systémových zdrojů při výpočetně náročných úlohách, tzv. *CPU bound tasks*. Úspěch se dostavil při horizontálním škálování, specificky při rozdělení databáze MongoDB mezi vícero uzlů. To vedlo ke zvýšení stability platformy a lepším výsledkům při zpracování velkých objemů dat. Rozsáhlá analýza a experimenty provedené V. Aranujou a spol. [1] odhalují, že ani populární a široce rozšířené platformy jako je platforma FIWARE nejsou dokonalé a že na poli systémů a platform pro zpracování dat z chytrých zařízení je stále prostor pro zlepšení a inovace.

4.1.2 ACADA (Asset Control and Data Acquisition)

Další z platform pro chytrá města je platforma ACADA (Asset Control and Data Acquisition) od české společnosti Logimic. Podle oficiální prezentace na webu společnosti Logimic [30] platforma umožňuje sběr a vyhodnocení dat týkajících se provozu zařízení, provádění lokálního i vzdáleného řízení zařízení, generování notifikací a alertů a zobrazení stavu a procesů zařízení na dashboardech. Oficiální prezentace dále uvádí, že platforma představuje plně konfigurovatelný systém s modulární architekturou, podporující mnoho komunikačních standardů. Platformu by mělo být možné snadno integrovat s technologiemi třetích stran a její provoz by neměl být náročný z pohledu hardwarových zdrojů.

Platforma data z chytrých zařízení nejprve ukládá do dvou typů databází: databáze časových řad InfluxDB a relační databáze PostgreSQL. Data jsou následně transformována na prakticky uplatnitelné poznatky v podobě vyhodnocení klíčových indikátorů výkonnosti. Ty platforma prezentuje uživatelům prostřednictvím webové aplikace. Vizualizaci vyhodnocení klíčových indikátorů výkonnosti v rámci webové aplikace platformy ukazuje obrázek 4.1.

⁴Zlom nastal když byla v dubnu roku 2019 v rámci Node.js 12 vydána funkce označovaná jako Worker Threads (https://nodejs.org/api/worker_threads.html), umožňující paralelní běh jazyka JavaScript.



Obrázek 4.1: Ukázka vizualizace vyhodnocení klíčových identifikátorů výkonnosti.⁵

Konkrétněji se jedná o 2 prvky uživatelského rozhraní. Každý z těchto prvků ukazuje, zdali hodnoty ze zařízení spadají do určitého rozmezí, či nikoliv. Platforma k reprezentaci rozsahu využívá dvojici klíčových indikátorů výkonnosti: první z nich stanovuje spodní mez, druhý horní. Vyhodnocení klíčových indikátorů výkonnosti reprezentuje barva: zelená barva reprezentuje kladné vyhodnocení a červená záporné.

Na zpracování dat v rámci platformy ACADA se ve své diplomové práci zaměřil O. Šulc [40]. Platforma je postavena na serverless architektuře a zpracování dat do podoby vyhodnocení klíčových indikátorů výkonnosti probíhá dávkově. Zpracování dat provádí periodicky spouštěná akce zvaná `watchAndProcessKpis`. Ta na základě dotazu do databáze získá definice všech klíčových indikátorů výkonnosti, které mají být vyhodnoceny. Každý z nich je pak vyhodnocen na základě relevantních dat ze senzorů, opět získaných z databáze. Tento proces měl značné výkonnostní nedostatky – zejména pak implementace akce `watchAndProcessKpis`. Vyhodnocení klíčových indikátorů výkonnosti probíhalo sekvenčně a komunikace s relační databází byla neefektivní. Výstupem práce O. Šulce [40] byla nová implementace akce `watchAndProcessKpis`, využívající vícero výpočetních vláken a komunikující s databází efektivně pomocí knihovny pro *objektově-relační mapování*, *ORM* z angl. *Object-relational Mapping*. Z pohledu času potřebného pro provedení akce tak došlo k až 27násobnému zrychlení⁶. I přes tyto přesvědčivé výsledky je v závěru práce uvedeno, že s rozvojem platformy a novými parametry, které bude třeba zpracovat, doba potřebná pro vyhodnocení klíčových indikátorů výkonnosti časem překročí přijatelnou mez. Práce proto doporučuje pozměnit způsob vyhodnocení klíčových indikátorů výkonnosti. Zdůrazňuje, že není nutné pokaždé vyhodnocovat všechny klíčové indikátory výkonnosti. Vyhodnocení různých typů klíčových indikátorů výkonnosti je třeba aktualizovat s různou frekvencí. Např. „infrastrukturní“ klíčové indikátory výkonnosti stačí aktualizovat jednou za hodinu a „environmentální“ jen jednou denně.

4.2 Požadavky na systém

Na základě analýzy uživatelských potřeb a rozboru existujících řešení lze konstatovat, že systém pro zpracování dat z chytrých zařízení by se měl zaměřit na následující:

⁵Jedná se o snímek obrazovky z uživatelského rozhraní webové aplikace platformy ACADA.

⁶O. Šulc [40] uvádí, že při využití všech čtyř jader CPU a zpracování 625 klíčových indikátorů výkonnosti se doba provádění akce `watchAndProcessKpis` snížila z původních 50 sekund na 1,8 sekundy.

- Proudové zpracování dat z chytrých zařízení v reálném čase – tento způsob zpracování dat představuje žádanou, ale přesto často chybějící vlastnost systémů a platforem z oblasti chytrých měst, průmyslu 4.0 atd. Umožňuje rychlejší zohlednění nových dat z chytrých zařízení a je tak předpokladem pro pružnější reakce personálu i vyšší efektivitu automatických procesů.
- Vysoká kapacita a škálovatelnost systému – je zřejmé, že s narůstajícím počtem chytrých zařízení bude potřeba zpracovat čím dál větší objem dat z těchto zařízení. Přibývat může také uživatelů. Má-li systém za těchto podmínek obstát, musí být navržen s důrazem na kapacitu a škálovatelnost.
 - Vysoká propustnost systému – u systémů zaměřených na proudové zpracování dat je kapacita systému dána zejména jeho propustností.
- Interoperabilita a rozšiřitelnost – systém musí být připraven na zpracování rozličných formátů dat, postupné zavádění dalších funkcí a integraci s jinými systémy.
- Stabilita – žádná z komponent systému nesmí nečekaně selhat.
- Srozumitelná reprezentace komplexní infrastruktury chytrých zařízení – primárním účelem systému je poskytovat uživatelům přehled o spravované infrastruktuře tvořené značným množstvím různých chytrých zařízení. Je nutné zvolit nástroj, který umožní transformaci velkého objemu surových dat z těchto zařízení na prakticky uplatnitelné poznatky s přidanou hodnotou jak pro uživatele, tak pro další systémy a aplikace.
 - Vhodné metody vizualizace – zvolenou podobu prakticky uplatnitelných poznatků je třeba vhodným způsobem vizualizovat a následně nabídnout uživatelům skrze přehledné uživatelské rozhraní systému.

Nad rámec výše uvedených požadavků na systém je třeba zohlednit ještě další, doplňující požadavky, které pochází od výzkumné skupiny zabývající se problematikou chytrých měst na Ústavu informačních systémů Fakulty informačních technologií a již zmiňované firmy Logimic. Jedná se o:

- Použití klíčových indikátorů výkonnosti – požadavek se váže na volbu nástroje určeného pro sledování chytrých zařízení a transformaci dat z těchto zařízení na prakticky uplatnitelné poznatky. Tím mají být, alespoň z hlediska terminologie, klíčové indikátory výkonnosti. O aplikovatelnosti klíčových indikátorů výkonnosti pro tento účel pojednává sekce 2.3.1. Podle této sekce mé práce, potažmo literatury, ze které jsem čerpal, se klíčové indikátory výkonnosti hodí pro reprezentaci stavu chytrých zařízení, snadnou interpretaci hodnot obsažených v datech z chytrých zařízení, identifikaci trendů a propojení dat s cíli uživatelů či strategickými cíli organizací. Splnění tohoto požadavku by tak nemělo představovat žádný problém.
- Řešení typu *proof of concept* – výsledkem mé práce by mělo být experimentální řešení typu proof of concept. To znamená, že namísto optimalizace existujícího a již delší dobu nasazeného řešení, jako je tomu v již zmiňované diplomové práci O. Šulce [40], je žádoucí tvorba úplně nového řešení, které sice nejspíš nebude možné obratem nasadit do reálného provozu, ale zato poslouží k ověření určitých hypotéz a otestování úspěšnosti určitých strategií a postupů.

- Použití technologie určené pro vývoj výkonných informačních systémů – jednou z hypotéz, kterou je žádoucí ověřit, je, že implementace systému pomocí některé z moderních technologií určených pro vývoj výkonných informačních systémů, kterým je věnována podkapitola 3.3, povede k vyšší výkonnosti systému a to zejména ve srovnání se současným řešením firmy Logimic, které je implementováno pomocí jazyka JavaScript a jeho běhového prostředí Node.js.
- Protokol MQTT – s ohledem na možnou pozdější integraci řešení do širšího ekosystému firmy Logimic je potřeba, aby systém podporoval komunikaci pomocí protokolu MQTT. To je nezbytné i proto, že testovací data z chytrých zařízení nasazených firmou Logimic budou dostupná skrze konkrétní téma touto společností nasazeného MQTT brokeru.
- Proprietární formát *Dev5* – pro příchozí MQTT zprávy bude použit proprietární formát firmy logimic označovaný jako Dev5. Ten je založen na populárním a široce používaném transportním formátu *JSON* (*JavaScript Object Notation*).
- Podpora pro ukládání dat z chytrých zařízení do databáze časových řad. Bude-li později potřeba data z chytrých zařízení ukládat do databáze časových řad, systém na to musí být připraven.

4.3 Případy užití systému

Systém bude uživatelům poskytovat přehled o infrastruktuře tvořené chytrými zařízeními. Tento přehled, založený na vizualizaci vyhodnocení patřičných klíčových indikátorů výkonnosti pro každé zařízení, si budou moci uživatelé kdykoliv zobrazit. To představuje první případ užití systému.

K vyhodnocení klíčových indikátorů výkonnosti dojde na základě jejich definic, které se v systému nemohou objevit, aniž by je do něj zadali uživatelé. Kromě tvorby nových definic by mělo být možné také zobrazit, upravovat a odstraňovat již existující definice. Tyto operace nad definicemi klíčových indikátorů výkonnosti tak představují další případy užití systému.

Vzhledem k tomu, že vyhodnocení klíčových indikátorů výkonnosti bude probíhat na základě příchozích MQTT zpráv, jsem provedl analýzu již zmiňovaného proprietárního formátu Dev5, který bude pro tyto zprávy použit. Obsah zprávy v tomto formátu se skládá z následujících položek:

- *unikátní identifikátor* (*UID*, z angl. *Unique IDentifier*) konkrétního chytrého zařízení,
- označení typu chytrého zařízení,
- páry klíč-hodnota, kde klíč představuje označení parametru daného typem chytrého zařízení a hodnota zase hodnotu parametru v rámci konkrétního zařízení.

To mě přivedlo k následujícím otázkám:

- Je třeba, aby systém zpracovával data ze všech typů chytrých zařízení komunikujících skrze dané téma MQTT brokeru?
- Je třeba u každého typu chytrého zařízení zohledňovat hodnoty všech parametrů?

Tyto otázky jsem konzultoval se zástupci již zmiňované výzkumné skupiny a bylo mi sděleno, že podmnožinu typů chytrých zařízení, které bude systém zohledňovat by měl definovat uživatel. Stejně tak by měl uživatel pro každý typ chytrých zařízení, který chce pomocí systému sledovat, definovat podmnožinu parametrů spojených s tímto typem chytrého zařízení, které považuje za relevantní. Tvorba definic relevantních typů chytrých zařízení zahrnujících výčet relevantních parametrů tak představuje další případy užití systému. Tyto definice by mělo být možné nejen tvořit, ale také zobrazit a upravit z hlediska relevantních parametrů.

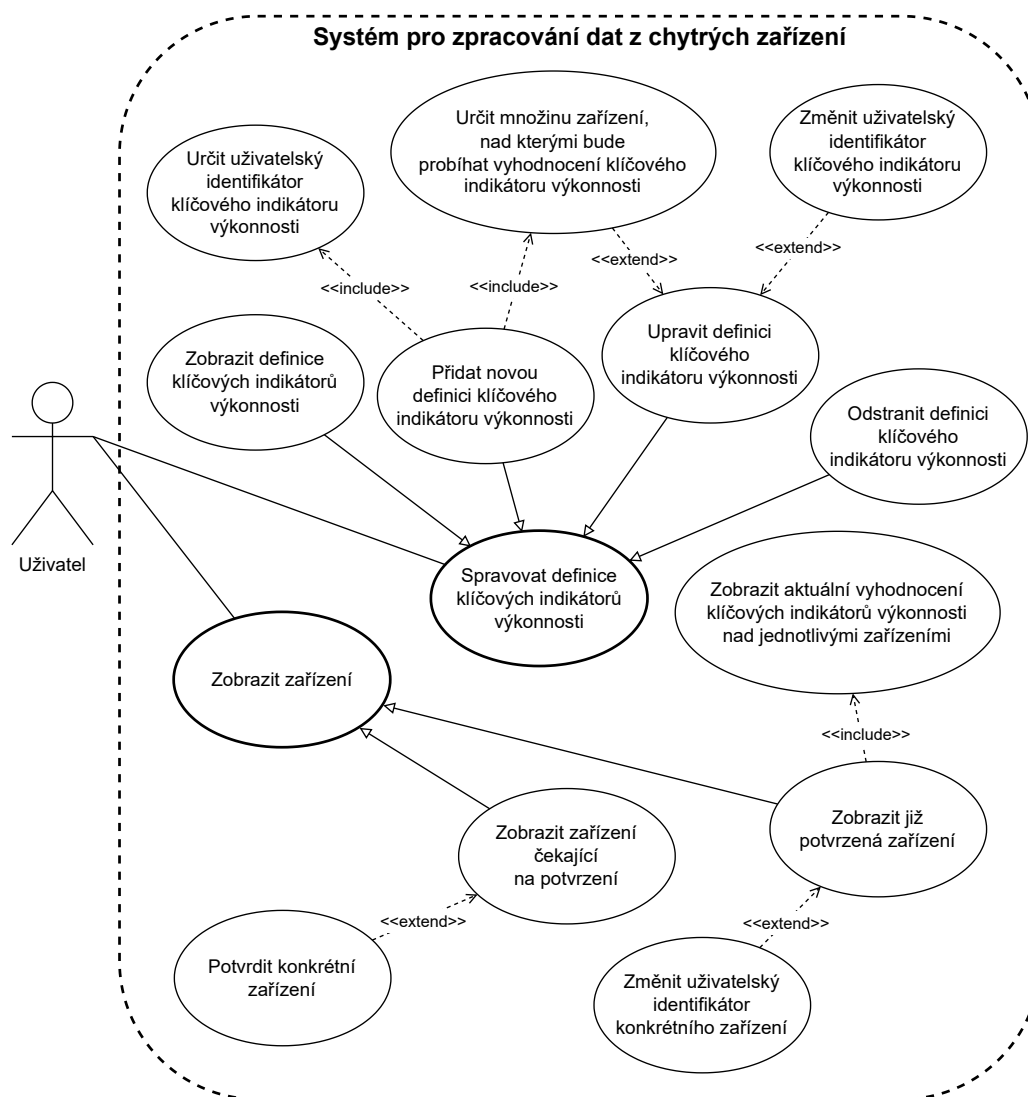
Následně jsem se zaměřil na uživatelskou přívětivost systému. Každé konkrétní zařízení bude v systému identifikováno pomocí unikátního identifikátoru. Uživatelé systému by ale nejspíš ocenili, kdyby bylo možné zařízení označit i jinak – způsobem, který pro ně bude uchopitelnější a přehlednější. Vhodný způsob označení představují uživatelské identifikátory – textové řetězce obsahující informace hodnotné pro uživatele. Zařízení ovšem nejsou jediným prvkem systému, u kterého je na místě zavést výše zmíněné uživatelské identifikátory. Uživatelským identifikátorem je třeba opatřit i klíčové indikátory výkonnosti. Tím bude zajištěno, že jejich význam z uživatelského hlediska bude ihned patrný a nebude jej třeba opakovaně vyvozovat z definice. Uživatelské identifikátory by mělo být možné také průběžně měnit, čímž bude zajištěna jejich kontinuální relevance.

Uživatelské identifikátory zařízení systém zpřehlední, ale pokud budou zařízení stovky či tisíce, nebude to stačit. Systém musí obsahovat mechanismus, pomocí kterého budou uživatelé moci jednotlivá chytrá zařízení seskupit. To zahrnuje tvorbu, editaci a odstraňování skupin zařízení, přičemž editace skupiny zařízení znamená, že do ní bude možné zařízení přidat a také z ní následně odebrat. I u skupin zařízení pak platí, že by je mělo být možné označit uživatelským identifikátorem, který půjde průběžně měnit.

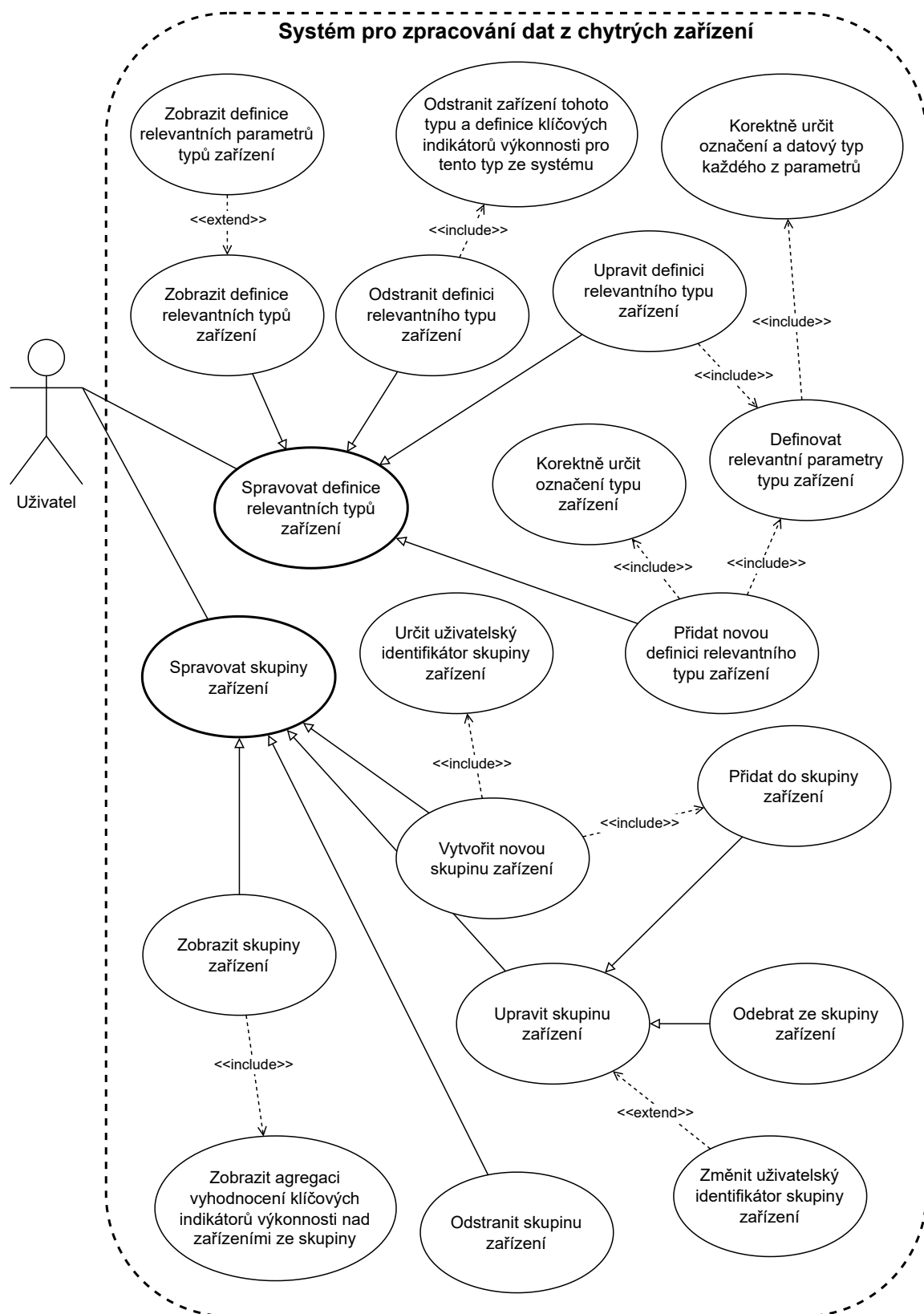
Co se týče definic klíčových indikátorů výkonnosti a jejich vyhodnocení, obdržel jsem následující instrukce:

- Každá z definic klíčových indikátorů výkonnosti by měla být spojena s konkrétní definicí relevantního typu zařízení – klíčový indikátor výkonnosti by měl být definován pro konkrétní typ chytrých zařízení.
- Uživatel by měl mít možnost určit, jestli bude daný klíčový indikátor výkonnosti vyhodnocován pro všechna zařízení daného typu v systému či pouze konkrétní podmnožinu těchto zařízení.
- Než bude možné zařízení přidat do skupiny zařízení či do množiny zařízení, nad kterými bude vyhodnocován klíčový indikátor výkonnosti definovaný pro typ zařízení, pod který zařízení spadá, je třeba, aby uživatel potvrdil, že má zájem s tímto zařízením pracovat. Než se tak stane, bude zařízení odděleno od zařízení, která již uživatelem potvrzena byla.

Identifikované případy užití systému zobrazuje diagram případů užití systému, který byl pro přehlednost rozdělen do dvou částí: 1. část zobrazuje obrázek 4.2 a 2. obrázek 4.3.



Obrázek 4.2: Diagram případů užití systému – 1. část



Obrázek 4.3: Diagram případů užití systému – 2. část

Kapitola 5

Návrh systému

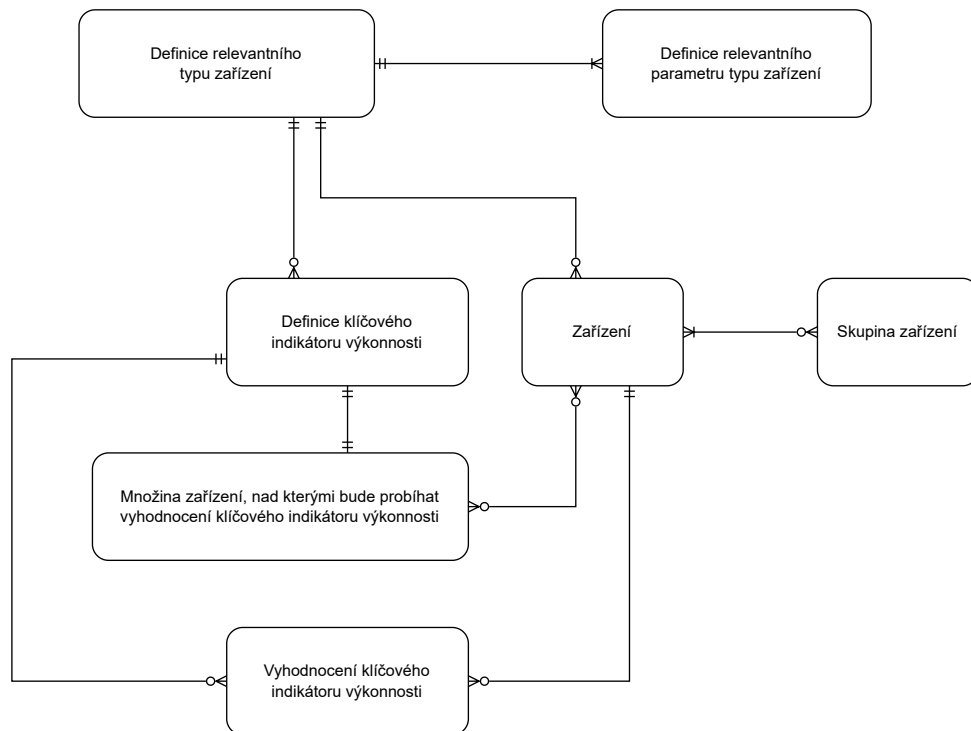
Při návrhu systému bylo třeba promyslet model ukládaných dat, specifikovat proces vyhodnocení klíčových indikátorů výkonnosti, následně vyřešit otázku stability systému, zvolit typ trvalého úložiště dat a nakonec navrhnout architekturu systému. To vše v souladu s poznatky a požadavky uvedenými v kapitole 4. Ve výše uvedeném pořadí jsem postupoval, neboť přesná specifikace modelu ukládaných dat byla prerekvizitou výběru vhodného typu trvalého úložiště dat a zvolený typ trvalého úložiště spolu se specifikací procesu vyhodnocení klíčových indikátorů výkonnosti a řešením otázky stability systému zase klíčovými vstupy pro návrh odpovídající architektury systému.

Z hlediska modelu ukládaných dat bylo žádoucí minimalizovat redundanci, tedy zajistit aby systém ukládal jen ta data, která skutečně ukládat musí – především pak definice, které do systému zadají uživatelé. Jedná se o:

- definice klíčových indikátorů výkonnosti,
- definice relevantních typů zařízení,
- definice skupin zařízení.

U definic klíčových indikátorů výkonnosti je třeba ukládat také „rozsah“ jejich vyhodnocení, tedy množinu zařízení, nad kterými mají být vyhodnocovány. Definice relevantních typů zařízení pak budou zahrnovat také definice relevantních parametrů daných typů zařízení – není totiž potřeba zohledňovat všechny parametry typů zařízení, viz podkapitola 4.3.

Systém bude ukládat také záznamy o jednotlivých chytrých zařízeních. Ty musí obsahovat příznak indikující, zdali již bylo zařízení potvrzeno uživatelem. Pro každé potvrzené zařízení pak bude systém ukládat poslední vyhodnocení patřičných klíčových indikátorů výkonnosti, tedy těch, které byly definovány pro typ daného zařízení a u kterých množina zařízení, nad kterými má docházet k vyhodnocení daného klíčového indikátoru výkonnosti, obsahuje dané zařízení. Vztahy mezi entitami představujícími ukládaná data zobrazuje diagram na obrázku 5.1.



Obrázek 5.1: Zjednodušený diagram vztahů mezi datovými entitami.

Systém bude moci obsahovat libovolné množství definic klíčových indikátorů výkonnosti, přičemž každá tato definice se bude vázat na jednu konkrétní definici relevantního typu zařízení. Definice relevantních typů zařízení pak budou zahrnovat definici alespoň jednoho relevantního parametru daného typu zařízení. Záznamy týkající se konkrétních zařízení se budou v systému objevovat na základě MQTT komunikace. Tato zařízení bude možné zařadit do libovolného počtu skupin zařízení a také libovolného počtu množin zařízení, nad kterými bude probíhat vyhodnocení konkrétních klíčových indikátorů výkonnosti. Smyslem skupin zařízení je zjednodušení interpretace výsledků zpracování dat z konkrétních zařízení, resp. vyhodnocení klíčových indikátorů výkonnosti nad konkrétními zařízeními, skrze agregaci. Smyslem množin zařízení, nad kterými bude probíhat vyhodnocení daného klíčového indikátoru výkonnosti, je omezení rozsahu vyhodnocování jednotlivých klíčových indikátorů výkonnosti. Důležité je také to, že vyhodnocení klíčových indikátorů výkonnosti se budou vždy vázat na jedno konkrétní zařízení.

5.1 Model klíčových indikátorů výkonnosti

Při návrhu modelu klíčových indikátorů výkonnosti jsem vycházel z formátu MQTT zpráv, na základě kterých bude probíhat jejich vyhodnocení, tedy již zmiňovaného formátu Dev5. Důležité jsou zejména parametry zařízení dané jeho typem a příslušné hodnoty těchto parametrů v rámci konkrétního zařízení. Každý z parametrů je jednoho z následujících tří datových typů:

- **number** – numerická hodnota,
- **string** – textový řetězec,

- **boolean** – binární pravdivostní hodnota.

Základem vyhodnocení klíčových indikátorů výkonnosti, a tedy i jejich modelu, byla transformace hodnot výše uvedených datových typů na hodnotu typu **boolean**:

- Z hodnoty typu **number** se stane hodnota typu **boolean** při aplikaci *binární relace* s mezní resp. referenční hodnotou, která bude také typu **number**, přičemž pojmem binární relace zde označuji rovnost či nerovnost. Podstatný je též rozdíl mezi ostrou a neostrou nerovností, spočívající v přístupu k referenční hodnotě.
- Hodnotu typu **string** lze na hodnotu typu **boolean** transformovat dvěma způsoby: na základě shodnosti s referenční hodnotou typu **string**, či abecedního porovnání s referenční hodnotou typu **string**. Vzhledem k povaze hodnot parametrů zařízení typu **string**, kterými jsou nejčastěji konstanty popisující konkrétní položku stavu zařízení, např. "on" či "off", je ale praktický přínos abecedního porovnání s referenční hodnotou značně omezený. Do modelu klíčových indikátorů výkonnosti jsem tedy zahrnul pouze shodnost hodnot typu **string**.
- U hodnot typu **boolean** je jedinou možností rovnost s referenční hodnotou typu **boolean**.

Kritéria pro převod hodnoty konkrétního parametru typu zařízení na hodnotu typu **boolean** představují „atomy“, ze kterých lze skládat složitější definice. To umožňují *logické operace* – pomocí logických operací, např. *konjunkce* ($\wedge$, **and**) a *disjunkce* ($\vee$, **or**), lze z několika hodnot typu **boolean** získat jednu. Transformace několika hodnot, tedy ne nutně 2, je umožněna díky *asociativitě* těchto binárních operací. Je-li binární operace asociativní, znamená to, že výsledek operace nezávisí na pořadí vyhodnocení dvojic operandů:

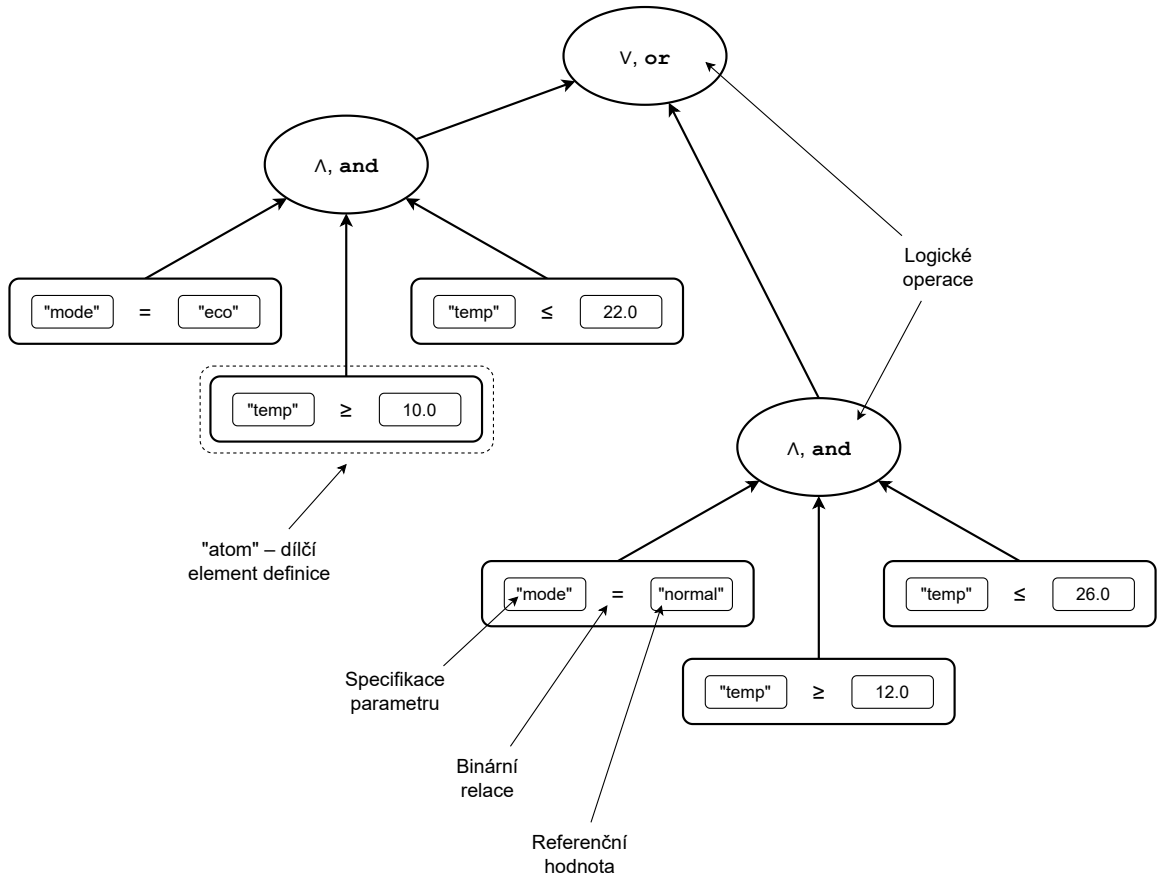
$$(A \wedge B) \wedge C \Leftrightarrow A \wedge (B \wedge C) \quad (5.1)$$

Kompletní sadu *Boolovy algebry*, pomocí které lze tvořit libovolné výrazy výrokové logiky představují operace konjunkce, disjunkce a *negace* ($\neg$, **not**). V počítači se logické výrazy, na kterých je model definic klíčových indikátorů výkonnosti založen, typicky reprezentují pomocí *binárních stromů*. Není to ale jediná možnost. Vzhledem k již zmíněné asociativitě logických operací lze použít také *n-ární stromy*, tedy stromy, jejichž uzly mají 0–n „potomků“. Obě datové struktury jsem tedy srovnal s důrazem na efektivitu vyhodnocení logických výrazů. Dospěl jsem k následujícím poznatkům:

1. Reprezentace pomocí binárních stromů – pro reprezentaci téhož výrazu je oproti n-árním stromům typicky potřeba více uzlů, což vede k hlubším, zanořenějším stromům. Tato skutečnost má značný dopad na složitost vyhodnocení výrazů – k vyhodnocení je potřeba více *rekursivních volání* či *iterací*. Výhodu z hlediska vyhodnocení představuje skutečnost, že pevně dané pořadí vyhodnocení operandů může být užitečné z hlediska určitých optimalizačních technik, např. tzv. *zkratového vyhodnocení*.
2. Reprezentace pomocí n-árních stromů – oproti binárním stromům je typicky potřeba méně úrovní, čímž se celá reprezentace výrazu stává kompaktnější. Menší množství uzlů, které je třeba vyhodnotit, může vést k rychlejšímu vyhodnocení celého výrazu, obzvláště, je-li zajištěno efektivní vyhodnocení na úrovni jednotlivých uzlů.

N-ární stromy také nabízí větší potenciál z hlediska paralelního vyhodnocení. Při vyhodnocování logických výrazů je ale přínos paralelního vyhodnocení značně omezen protívahou přínosu zkratového vyhodnocení, přičemž tyto koncepty jsou jen těžko slučitelné. Posledním argumentem je náročnost aplikace optimalizačních technik, která je u n-árních stromů o něco vyšší.

Z výše uvedených poznatků vyplývá, že nelze jednoznačně určit, která z datových struktur je pro reprezentaci definic klíčových indikátorů výkonnosti na bázi logických výrazů vhodnější. S přihlédnutím k flexibilitě, kterou poskytují, a typicky menší míře zanoření jsem se nakonec rozhodl pro reprezentaci logických výrazů představujících definice klíčových indikátorů výkonnosti pomocí n-árních stromů. Negaci jsem nahradil *negovanou disjunkcí* ($\downarrow$, **nor**), která je binární, potažmo n-ární a při aplikaci na jeden operand je obdobou negace. Příklad definice konkrétního klíčového indikátoru výkonnosti odpovídající výše popsámu modelu ilustruje obrázek 5.2.



Obrázek 5.2: Příklad definice klíčového indikátoru výkonnosti.

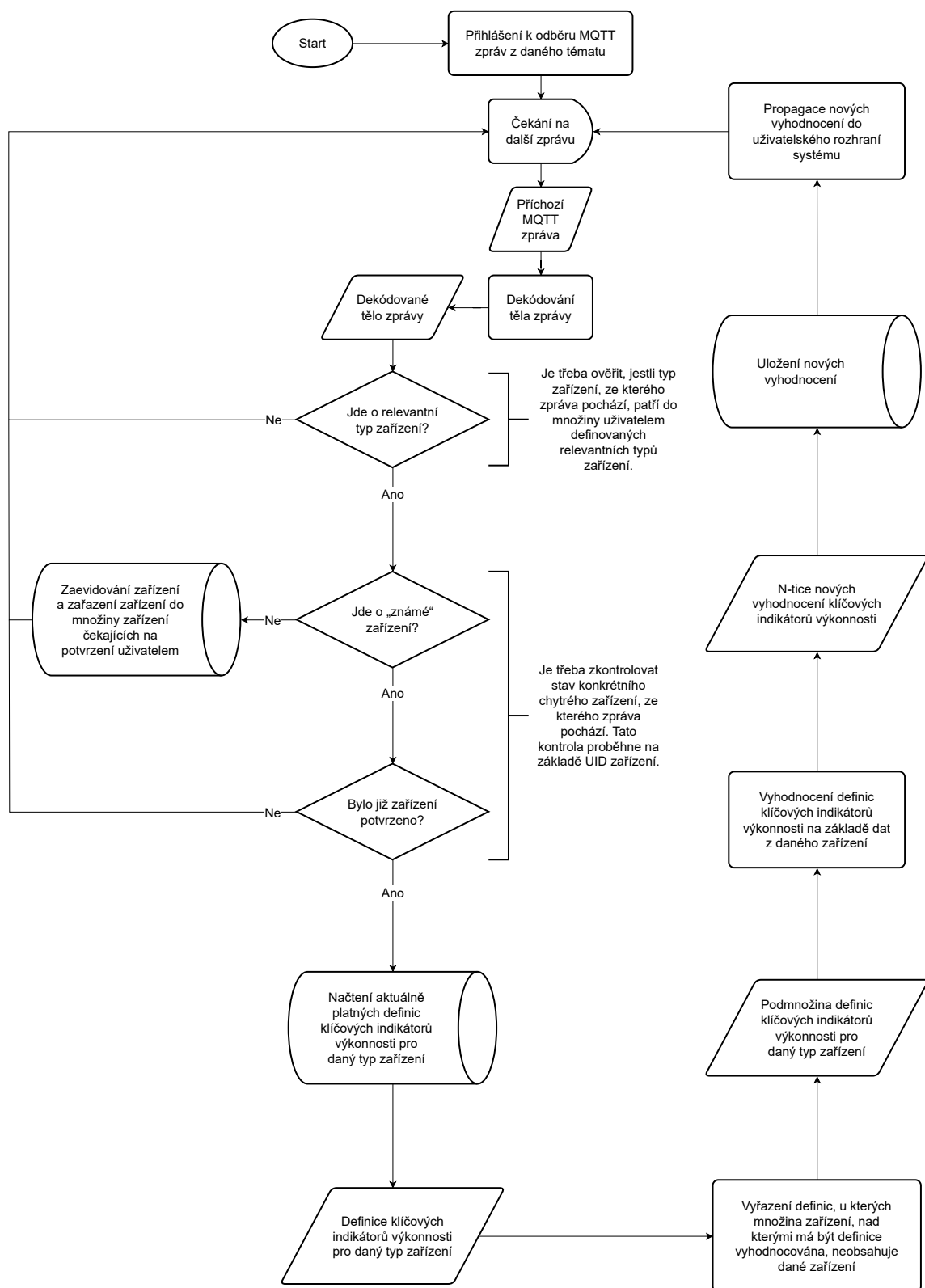
Smyslem tohoto klíčového indikátoru výkonnosti je kontrola, zdali teplota zařízení (hodnota parametru "temp") spadá do přijatelného rozmezí, daného režimem zařízení (hodnota parametru "mode"). V režimu "eco" je přijatelné rozmezí teploty zařízení od 10 do 22 °C. V režimu "normal" pak od 12 do 26 °C.

5.2 Vyhodnocení klíčových ukazatelů výkonnosti

Vyhodnocení klíčových indikátorů výkonnosti bude probíhat průběžně na základě příchozích zpráv, odebíraných z MQTT tématu. U každé zprávy je třeba dekodovat tělo zprávy a ověřit, jestli typ zařízení, ze kterého zpráva pochází, patří do množiny uživatelem definovaných relevantních typů zařízení. Pokud ne, systém se nebude zprávou dále zabývat. Pokud ano, je třeba zkontrolovat stav konkrétního chytrého zařízení, ze kterého zpráva pochází. Existují tři scénáře:

1. Neznámé zařízení – jedná-li se o první zprávu z konkrétního zařízení, systém toto zařízení zaeviduje a zařadí do množiny zařízení čekajících na potvrzení uživatelem. K vyhodnocení klíčových indikátorů výkonnosti nedojde.
2. Známé zařízení, které zatím nebylo potvrzeno – jedná-li se o zprávu ze zařízení, které systém již eviduje, ale zatím nebylo potvrzeno ze strany uživatele, nemůže dojít k vyhodnocení klíčových indikátorů výkonnosti a systém se nebude zprávou dále zabývat.
3. Známé, již potvrzené zařízení – pochází-li zpráva ze zařízení, které systém eviduje a které již bylo potvrzeno uživatelem, proběhne vyhodnocení klíčových indikátorů výkonnosti.

Proces vyhodnocení klíčových indikátorů výkonnosti začne načtením aktuálně platných definic pro daný typ zařízení. U každé z definic je následně třeba ověřit, zdali dané zařízení náleží do množiny zařízení, nad kterými má být daná definice vyhodnocována. Pokud ano, proběhne vyhodnocení definice. Pokud ne, definice bude vynechána. Výstupem procesu je n-tice nových vyhodnocení klíčových indikátorů výkonnosti nad daným konkrétním zařízením. Tato vyhodnocení je třeba uložit a také je propagovat do uživatelského rozhraní systému. Výše popsany proces vyhodnocení klíčových ukazatelů výkonnosti přehledně zobrazuje vývojový diagram v rámci obrázku 5.3.



5.3 Stabilita a propustnost systému

Důležitými požadavky uvedenými v kapitole 4 jsou proudové zpracování dat a stabilita systému. Stabilní systém je systém, u kterého nedochází k selháním jednotlivých komponent či celého systému díky přetížení, tedy nadměrné zátěži. Podle knihy M. Harchol-Balterové [17], ze které jsem čerpal informace ohledně teorie front, je stabilita systémů provádějících proudové zpracování dat otázkou splnění následující podmínky:

$$\lambda < \mu \quad (5.2)$$

Symbol λ zde značí *frekvenci příchodů*, z angl. *arrival rate*, a μ *frekvenci obsluhy*, z angl. *service rate*. Je-li frekvence obsluhy vyšší než frekvence příchodů, znamená to, že systém dokáže obsloužit příchozí prvky či úlohy, aniž by docházelo k zahlcení. Lze tedy konstatovat, že předpokladem pro stabilitu systému je jeho dostatečná propustnost. Stability systému se týká i tzv. *Littleův zákon*, z angl. *Little's Law*, který představuje následující rovnice:

$$L = \lambda \times W \quad (5.3)$$

Průměrný počet prvků v systému (L) je dán součinem frekvence příchodů (λ) a *průměrné doby, po kterou se prvek v systému nachází* (W). Z hlediska stability systému je žádoucí udržet průměrný počet prvků systému pod určitou mezí, čehož lze dosáhnout buďto omezením frekvence příchodů, omezením doby, po kterou se prvek nachází v systému a nebo kombinací obou přístupů. Vzhledem k tomu, že dalším z požadavků na systém pro zpracování dat z chytrých zařízení je co nejrychlejší dostupnost poznatků založených na posledních datech ze zařízení, je zřejmé, že omezení frekvence příchodů (λ) není optimálním řešením. Jako vhodnější se jeví minimalizace doby, po kterou se zpráva ze zařízení bude nacházet v systému (W). Na základě následující rovnice, platné pro *model systémů s jednou jednotkou obsluhy*, z angl. *server*, označovaný jako $M/M/1$ ¹ je zřejmé, že jedním z klíčů pro minimalizaci doby, po kterou se zpráva ze zařízení bude nacházet v systému (W), je co nejvyšší frekvence obsluhy (μ).

$$W = \frac{1}{\mu - \lambda} \quad (5.4)$$

Frekvenci obsluhy (μ) v systémech odpovídajících modelu $M/M/1$ ale nelze navyšovat neomezeně. Při vysoké frekvenci příchodů (λ) nemusí být možné udržet přijatelnou hodnotu doby, po kterou se zpráva ze zařízení bude nacházet v systému (W), což může ohrozit stabilitu systému. Řešením je přidání dalších jednotek obsluhy a zavedení paralelní obsluhy – adopce modelu $M/M/k$ ². Celkovou frekvenci obsluhy v rámci modelu $M/M/k$ lze určit pomocí následující rovnice:

$$\mu = k \times \mu_i \quad (5.5)$$

Proměnná k zde značí počet jednotek obsluhy a μ_i dílčí frekvenci obsluhy jedné jednotky. Klíčem k zajištění stability systému odpovídajícího modelu $M/M/k$ je tedy dostatečná frekvence obsluhy jedné jednotky v kombinaci s dostatečným počtem jednotek umožňujících paralelní obsluhu.

¹Jedná se o tzv. *Kendallovu notaci*, z angl. *Kendall's notation*.

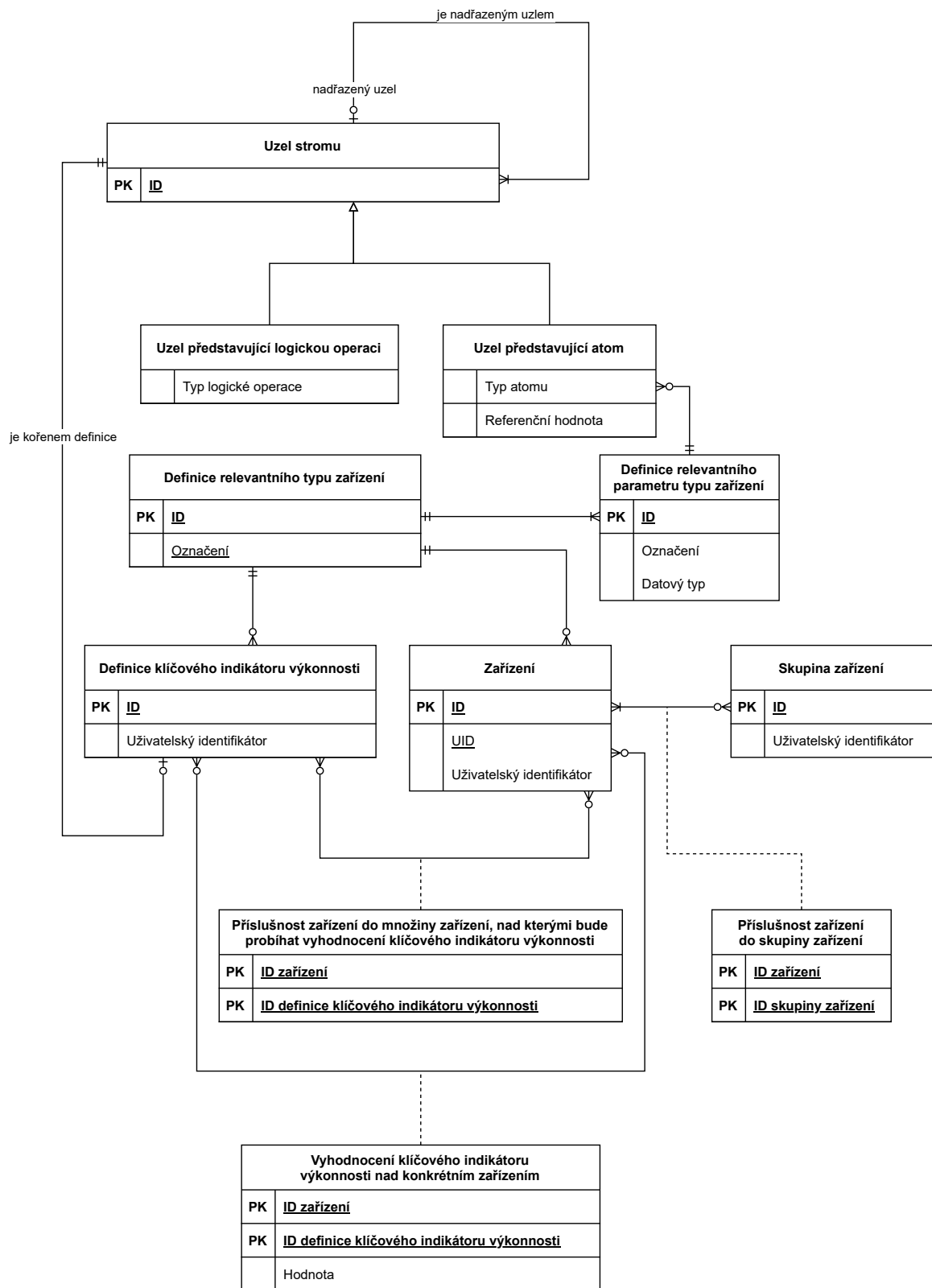
²Tento model se označuje také jako model $M/M/c$ či model *Erlang-C*.

Praktické implikace

Výše uvedená teorie má zřejmé praktické implikace – dostatečnou propustnost systému pro zpracování dat z chytrých zařízení lze zajistit dostatečně rychlým zpracováním jedné zprávy v kombinaci s dostatečným počtem kanálů pro paralelní zpracování zpráv. Co se týče minimalizace doby potřebné pro zpracování jedné zprávy, důležitou roli hraje co nejrychlejší „zahození“ zcela irrelevantních zpráv, tedy zpráv pocházejících ze zařízení, jejichž typ nepatří do množiny relevantních typů chytrých zařízení definovaných uživatelem. Dále pak rychlé ověření stavu konkrétního zařízení, viz vývojový digram v rámci obrázku 5.3. Množina relevantních typů zařízení a záznamy o konkrétních zařízeních, která již systém eviduje, by tak měly být přístupné co nejsnáze. Významný dopad na dobu potřebnou pro zpracování jedné zprávy bude mít také vyhodnocení definic klíčových indikátorů výkonnosti pro daný typ zařízení, které by mělo proběhnout paralelně, a uložení výsledků vyhodnocení spolu s jejich propagací do uživatelského rozhraní systému. Při paralelním zpracování zpráv z chytrých zařízení je také potřeba zabránit kolizím a zajistit tak konzistenci a integritu sdílených dat. To usnadní centralizace přístupu k trvalému úložišti dat.

5.4 Trvalé úložiště dat – relační databáze

Volba typu trvalého úložiště dat by měla být založena především na povaze ukládaných dat, resp. modelu ukládaných dat, režimu přístupu k trvalému úložišti dat a na požadavcích týkajících se výkonnosti trvalého úložiště dat a konzistence ukládaných dat. Ukládaná data jsou strukturovaná a mezi jednotlivými entitami jsou jasně dané vazby – to ukazuje na *relační model* dat. Výjimku představuje *hierarchický model* definic klíčových indikátorů výkonnosti, založený na n-árních stromech. Lze předpokládat, že nejčastějšími operacemi zahrnujícími trvalé úložiště dat budou operace, ke kterým bude docházet na základě zpracování příchozích zpráv z chytrých zařízení, tedy především operace zápisu. Vzhledem k tomu, že data v systému mají dlouhodobě co nejpřesněji reflektovat stav chytrých zařízení, je důležitá i konzistence dat. Operace nad trvalým úložištěm dat by měly mít podobu spolehlivých *transakcí*. S ohledem na strukturu modelu ukládaných dat, potřebu udržet konzistenci dat a další skutečnosti jsem se rozhodl pro relační databázi. Stromové struktury představující definice klíčových indikátorů výkonnosti bylo možné ukládat buďto do druhé databáze odlišného typu a nebo zajistit ukládání těchto struktur pomocí tabulek relační databáze. Rozhodl jsem se pro druhou možnost – ukládat celý model do jedné databáze je snazší a stromové struktury představující definice klíčových indikátorů výkonnosti pravděpodobně nebudou dosahovat takové hloubky, resp. úrovně zanoření, aby jejich ukládání do relační databáze mělo výrazný dopad na výkonnost systému. Výsledný konceptuální model relační databáze zahrnující ukládání definic klíčových indikátorů výkonnosti představuje ER diagram na obrázku 5.4, který vychází z dříve uvedeného zjednodušeného diagramu vztahů mezi datovými entitami na obrázku 5.1.



Obrázek 5.4: ER diagram – konceptuální model relační databáze.

5.5 Architektura systému

Jako první jsem uvažoval o moderní a z provozně-ekonomického hlediska atraktivní serverless architektuře popsané v sekci 3.1.5. Tu jsem ale posléze zavrhl s ohledem na mé omezené zkušenosti s touto architekturou a obavy ze situace označované jako *proprietární uzamčení*, z angl. *vendor lock-in*. Proprietární uzamčení spočívá v tom, že návrh systému je úzce provázán s prostředím konkrétního poskytovatele cloudových služeb a systém tak nelze nasadit a provozovat mimo rámec daný infrastrukturou a službami tohoto poskytovatele. V konečném důsledku je tak firma či organizace, která potřebuje systém provozovat, „vydána na milost“ cenovým tarifům a dalším podmínkám poskytovatele.

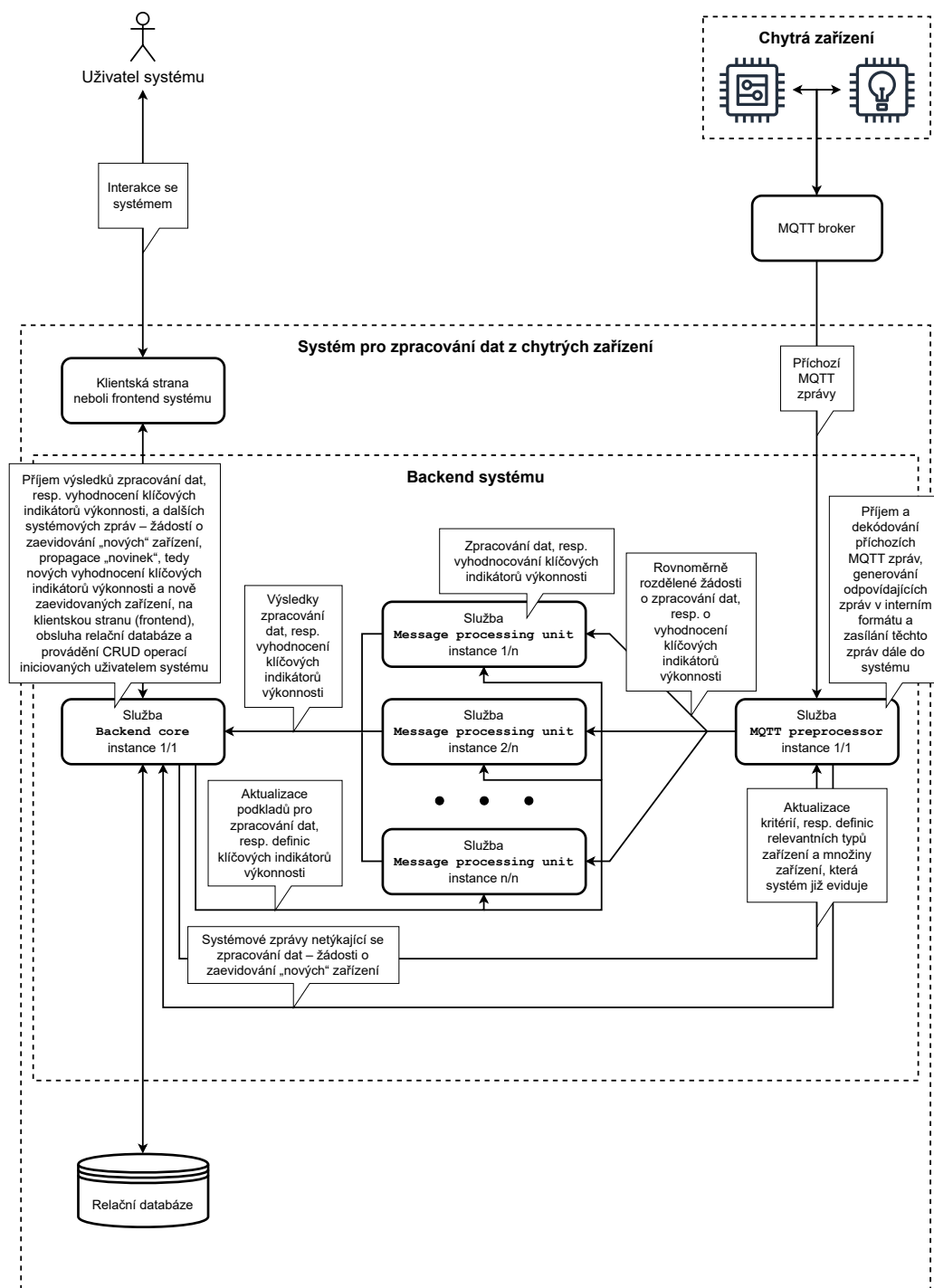
Snáze uchopitelnou alternativu představovala tříúrovňová architektura popsaná v sekci 3.1.2 – klientská strana, neboli frontend, zajišťující interakci uživatelů se systémem, backend provádějící proudové zpracování dat a relační databáze sloužící pro trvalé uložení patřičných dat. Standardní interpretace této architektury ale směřuje na monolitický backend, což by znamenalo, že kapacita systému by byla omezena systémovými zdroji stroje, na kterém by backend systému běžel. Jinými slovy – systém by bylo možné škálovat pouze vertikálně. Tento problém řeší architektura založená na mikroslužbách popsaná v sekci 3.1.4. Rozhodl jsem se tedy touto architekturou inspirovat a backend systému rozdělil do trojice služeb. Systém tak půjde do určité míry škálovat i horizontálně a díky rozdělení zdrojového kódu mezi služby jej bude možné lépe udržovat a rozvíjet. Jedná se o následující služby:

- První služba – **MQTT preprocessor**, bude přijímat příchozí MQTT zprávy, dekodovat těla těchto zpráv a na základě jejich obsahu a dalších kritérií generovat odpovídající zprávy v interním formátu, které budou zasílány dále do systému. Generovanými zprávami budou žádosti o zaevidování „nových“ zařízení a žádosti o vyhodnocení klíčových indikátorů výkonnosti na základě dat obsažených v původní MQTT zprávě z konkrétního zařízení.
- Druhá služba – **Message processing unit**, bude provádět samotné zpracování dat, resp. vyhodnocení klíčových indikátorů výkonnosti, a to na základě explicitních žádostí zaslaných první službou.
- Třetí služba – **Backend core**, bude zodpovídat za zbylou funkcionalitu backendu systému. To zahrnuje příjem výsledků zpracování dat, resp. vyhodnocení klíčových indikátorů výkonnosti, a dalších systémových zpráv, tedy žádostí o zaevidování „nových“ zařízení, propagaci „novinek“, tedy nových vyhodnocení klíčových indikátorů výkonnosti a nově zaevidovaných zařízení, na klientskou stranu (frontend), obsluhu relační databáze a provádění *CRUD*³ operací iniciovaných uživatelem systému.

Výše zmíněné horizontální škálování se bude týkat druhé služby – **Message processing unit**. Škálovatelnost služby **MQTT preprocessor** omezuje potřeba kontrolovaného příjmu MQTT zpráv zatímco u služby **Backend core** horizontálnímu škálování brání, jinak žádoucí, centralizovaný přístup k relační databázi, usnadňující zajištění konzistence a integrity dat. Tato architektura, kterou přehledně zobrazuje diagram v rámci obrázku 5.5, tak představuje kompromis, přičemž jedním cílem je zvýšení propustnosti systému pomocí horizontálního škálování a druhým, protichůdným cílem, je omezení komplexity a

³Zkratkou *CRUD* se označuje typická čtveřice operací nad daty v informačních systémech: *tvorba* – *Create*, *čtení* – *Read*, *aktualizace* – *Update* a *mazání* – *Delete*.

rizik spojených s distribuovanými systémy. Za zmínku stojí i rozšiřitelnost systému daná touto architekturou – pro integraci dalšího zdroje dat z chytrých zařízení by stačilo přidat službu analogickou ke službě MQTT preprocessor. Zbytek systému by pak mohl zůstat beze změny. Tento mechanismus je jistou obdobou komponent již mnohokrát zmiňované platformy FIWARE, označovaných jako IoT agenti, které zprostředkovávají komunikaci mezi chytrými zařízeními a platformou.



Obrázek 5.5: Diagram navržené architektury systému.

Kapitola 6

Implementace systému

Systém byl implementován jako plnohodnotný, samostatný software, přičemž kompletní zdrojový kód je sdružen do jednoho repositáře – jedná se o tzv. *monorepo*. Implementace různých částí systému proběhla pomocí různých technologií s důrazem na rozdílné cíle. Zatímco frontend systému, zahrnující uživatelské rozhraní, byl implementován pomocí webových technologií, což umožňuje práci se systémem na počítačích, chytrých telefonech a dalších zařízeních disponujících webovým prohlížečem, backend systému byl implementován pomocí programovacího jazyka Go, který exceluje z hlediska podpory souběžnosti a paralelismu, s důrazem na výkonnost a efektivitu. Důležitou roli hraje i aplikační rozhraní na bázi technologie GraphQL¹. Toto rozhraní poskytuje „oficiálnímu frontendu“², ale i dalším potenciálním klientům systému, flexibilitu z hlediska dotazů na data i jejich modifikace a backendu zase možnost zasílat na klientskou stranu notifikace bez neustálých dotazů ze strany klientů. Služby tvořící backend systému mezi sebou komunikují pomocí technologie RabbitMQ³, primárně založené na již zmiňovaném protokolu AMQP, viz sekce 2.1.1. Běh systémových služeb, zahrnujících podpůrné služby, zajišťuje platforma Docker⁴, umožňující spolehlivý běh aplikací v odlehčeném, izolovaném prostředí, které se označuje pojmem *kontejner*, z angl. *container*. Nad touto platformou pak operuje ještě nástroj Docker Compose⁵, zaměřený na spouštění multi-kontejnerových aplikací, mezi které systém patří.

6.1 Klientská strana a uživatelské rozhraní

Implementace klientské strany systému proběhla pomocí webových technologií, přesněji pomocí programovacího jazyka TypeScript⁶ a knihovny pro tvorbu uživatelského rozhraní React⁷. Uživatelské rozhraní systému je tak sestaveno z mnoha opakovaně použitelných komponent, což usnadňuje jeho údržbu i rozvoj. Přínosem knihovny React je i deklarativní přístup k tvorbě uživatelského rozhraní. Programátor se nemusí zabývat přímou manipulací s *objektovým modelem dokumentu*, *DOM* z angl. *Document Object Model*, a namísto toho

¹Viz <https://graphql.org/>.

²Spojení „oficiální frontend“ označuje klientskou stranu systému, neboli frontend, s důrazem na to, že tato klientská strana je přímou součástí implementace systému. Nemusí se ale jednat o jediného klienta systému. Tzn. k systému, resp. k backendu systému, se mohou jako klienti přihlásit také různé další systémy a aplikace.

³Viz <https://www.rabbitmq.com/>.

⁴Viz <https://www.docker.com/>.

⁵Viz <https://docs.docker.com/compose/>.

⁶Viz <https://www.typescriptlang.org/>.

⁷Viz <https://react.dev/>.

pouze určí, jak mají prvky uživatelského rozhraní „vypadat“ v závislosti na jejich stavu. Při změně stavu aplikace dochází k automatickému překreslení nezbytné části uživatelského rozhraní, což zvyšuje responzivitu systému. Použití jazyka TypeScript se silným a statickým systémem typů pro logiku klientské strany systému činí tutu část systému spolehlivější, neboť značné množství chyb je zachyceno již při překladu. Vzhled prvků uživatelského rozhraní určují aplikované *kaskádové styly*, *CSS* z angl. *Cascading Style Sheets*, přesněji pak CSS rámec Tailwind CSS⁸. Tento rámec představuje určitou změnu v přístupu ke kaskádovým stylům – namísto konkrétních, „sémantických“ tříd definovaných v oddělených souborech, tzv. *stylesheets*, se používají obecnější, „pomocné“ třídy, které lze zahrnout přímo do .tsx souborů jednotlivých komponent. Výsledkem jsou adresné a transparentní a definice vzhledu prvků uživatelského rozhraní, které lze snadno měnit. Tvorbu a kalibraci uživatelského rozhraní systému jsem si dále usnadnil použitím nástroje Parcel⁹, sloužícího k sestavení projektu, s nízkými nároky na konfiguraci, a MUI¹⁰, sady předdefinovaných komponent knihovny React. Z této sady jsem použil komponenty realizující elementární prvky uživatelského rozhraní: textová pole, zaškrtačovací pole, rozbalovací seznamy, modální okna atd. To mi umožnilo soustředit se na kompozici uživatelského rozhraní na úrovni sekcí a obrazovek a také logiku obsluhující uživatelské interakce a komunikaci s backendem systému.

6.1.1 Editor definic klíčových indikátorů výkonnosti

Dobrým příkladem uživatelského rozhraní systému je editor definic klíčových indikátorů výkonnosti, který umožňuje intuitivní tvorbu a úpravu definic dle modelu popsaného v podkapitole 5.1. Primárním prvkem uživatelského rozhraní je zde interaktivní editor stromových struktur, jehož implementace se opírá o komponentu React D3 Tree¹¹, založenou na již zmiňované knihovně D3, sloužící pro vizualizaci dat pomocí jazyka JavaScript. Tato komponenta je nicméně založena na *škálovatelné vektorové grafice*, *SVG*, z angl. *Scalable Vector Graphics*. Abych mohl pro jednotlivé uzly stromu použít komponenty knihovny React, použil jsem SVG značku `<foreignObject>`. Při kliknutí na uzel konkrétního typu se uživateli zobrazí modální okno¹², vybízející uživatele k interakci. V případě uzlu reprezentujícího „atom“ je třeba zvolit parametr zařízení daný jeho typem, binární relaci a referenční hodnotu. V případě uzlu reprezentujícího logickou operaci pak její typ: konjunkci, disjunkci, či negovanou disjunkci. Klikne-li uživatel na imaginární uzel reprezentující možnost přidat na daném místě nový uzel, označený symbolem +, je nejprve vyzván, aby zvolil jestli má nový uzel reprezentovat „atom“ či logickou operaci a následně aby uzel zvoleného typu nakonfiguroval.

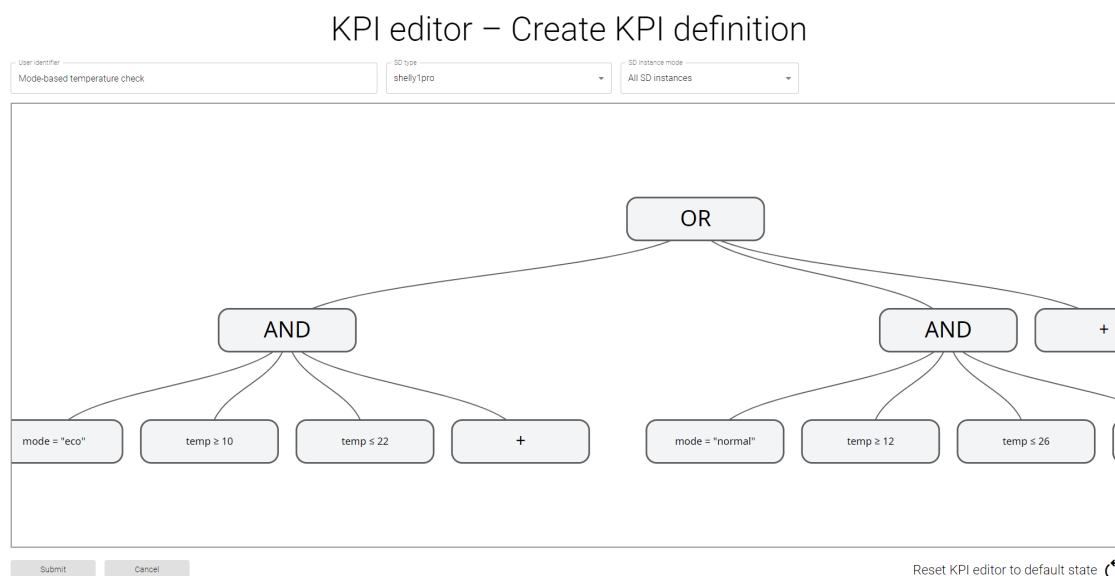
⁸Viz <https://tailwindcss.com/>.

⁹Viz <https://parceljs.org/>.

¹⁰Viz <https://mui.com/>.

¹¹Viz <https://www.npmjs.com/package/react-d3-tree>.

¹²Pro snazší správu modálních oken napříč aplikací jsem využil balíček Nice Modal (viz <https://www.npmjs.com/package/@ebay/nice-modal-react>) z open-source kolekce společnosti eBay.



Obrázek 6.1: Editor definic klíčových indikátorů výkonnosti.

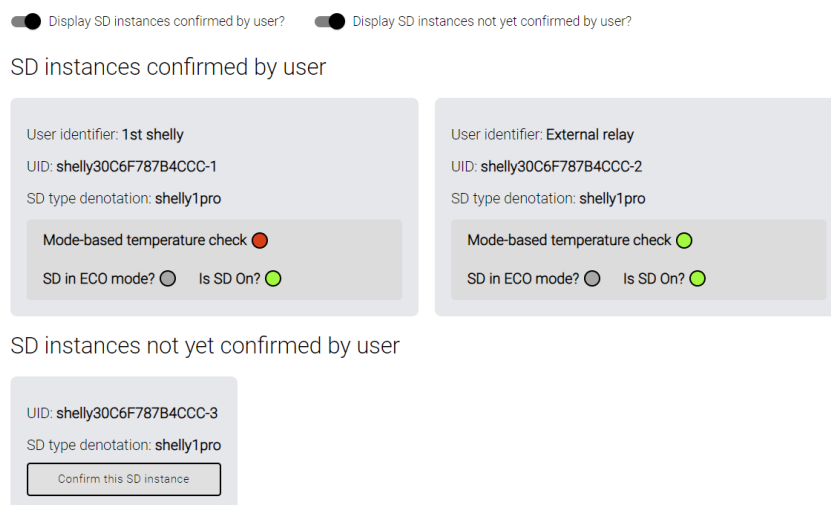
Na obrázku 6.1 byla skrze editor vytvořena definice, uvedená jako praktický příklad modelu popsaného v podkapitole 5.1, kterou ukazuje obrázek 5.2. Značná vizuální shoda mezi ilustrací modelu klíčových indikátorů výkonnosti na obrázku 5.2 a snímkem obrazovky z uživatelského rozhraní systému zachycujícím editor definic na obrázku 6.1 je výsledkem snahy poskytnout uživateli nezkreslenou představu o modelu definic klíčových indikátorů výkonnosti a editor prostřednictvím kterého bude možné využít plný potenciál tohoto modelu. Do editoru lze přejít z obrazovky „KPI definitions“, zobrazující přehled definic klíčových indikátorů výkonnosti, kliknutím na ikonu tužky¹³ na kartě některé z definic, a nebo kliknutím na kartu se symbolem +. V prvním případě jde o úpravu zvolené definice, zatímco v druhém o tvorbu nové definice.

6.1.2 Sledování chytrých zařízení

Primárním účelem systému je, dle jednoho z bodů podkapitoly 4.2, poskytnout uživateli přehled infrastruktury tvořené mnoha chytrými zařízeními. Tuto úlohu plní obrazovka „SD instances“, přičemž zkratka SD označuje chytrá zařízení, z angl. *Smart Device*, a pojem instance zdůrazňuje, že se jedná o konkrétní chytrá zařízení. Obsahem této obrazovky uživatelského rozhraní systému, kterou ukazuje obrázek 6.2, jsou karty konkrétních chytrých zařízení rozdělené do dvou sekcí: chytrá zařízení která již byla potvrzena uživatelem a chytrá zařízení, k jejichž potvrzení uživatelem zatím nedošlo. Karty již potvrzených a zatím nepotvrzených zařízení se liší i obsahem: zatímco u zařízení, která již byla potvrzena uživatelem, se zobrazuje textové pole, jehož obsahem je uživatelský identifikátor daného zařízení, na kartě zařízení, které ještě potvrzeno nebylo, toto pole nahrazuje tlačítko umožňující uživateli dané zařízení potvrdit.

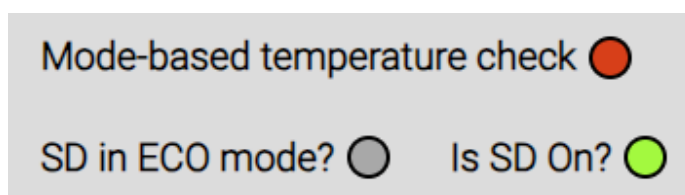
¹³Tužka je standardně používaným symbolem pro operace zahrnující úpravy již existujících prvků či záznamů.

SD instances



Obrázek 6.2: Obrazovka „SD instances“.

Nejvýznamnějším prvkem uživatelského rozhraní, který se nachází výhradně na kartách již potvrzených chytrých zařízení, je sekce zobrazující poslední vyhodnocení klíčových indikátorů výkonnosti nad daným chytrým zařízením, resp. daty z tohoto zařízení. Tato sekce zobrazuje páry tvořené uživatelským identifikátorem klíčového indikátoru výkonnosti a jednoduchým grafickým prvkem reprezentujícím výsledek vyhodnocení klíčového indikátoru výkonnosti. Zde jsem se inspiroval sémantickým kódováním pomocí barev, o kterém pojednává publikace L. Pappasové a spol. [32], a také platformou ACADA od firmy Logimic, kterou popisuje sekce 4.1.2 – kladné vyhodnocení značí zelená barva a záporné červená. Situaci, kdy u konkrétního zařízení k vyhodnocení klíčového indikátoru výkonnosti doposud nedošlo, pak značí šedá barva. Příklad obsahu výše popsané sekce na kartě konkrétního zařízení ukazuje obrázek 6.3.



Obrázek 6.3: Sekce zobrazující vyhodnocení klíčových indikátorů výkonnosti.

Stejná sekce se nachází i na kartách skupin zařízení, v rámci obrazovky „SD instance groups“. Zde červená barva značí, že alespoň u jednoho zařízení ze skupiny je vyhodnocení daného klíčového indikátoru výkonnosti záporné, zatímco zelená, že u všech zařízení ze skupiny je vyhodnocení kladné, či zatím nedostupné. Šedá barva v tomto případě signalizuje, že vyhodnocení zatím neproběhlo u žádného zařízení ze skupiny.

6.2 Aplikační rozhraní na bázi technologie GraphQL

Aplikační rozhraní systému, resp. služby **Backend core**, je založeno na technologii GraphQL. Tato technologie zahrnuje specifikaci dotazovacího jazyka, systém typů, *introspekcí* a tzv. *execution engine* – komponentu, která implementuje abstraktní logiku zpracování požadavků směřujících na aplikační rozhraní. Má-li toto aplikační rozhraní skutečně fungovat, je třeba definovat tzv. *schéma*, tedy strukturu dat, za pomoci systému typů GraphQL, a také operace aplikačního rozhraní: operace čtení, tzv. *queries*, a operace sloužící k modifikaci dat, tzv. *mutations*. Dále je nutné v rámci backendu implementovat tzv. *resolvers* či *resolver functions* – speciální funkce, které určují jak bude daná operace, tedy čtení či modifikace dat, provedena. GraphQL schéma systému obsahuje soubor `schema.graphqls`, zatímco implementace resolver functions se nachází v souboru `schema.resolvers.go`. Oba zmíněné soubory se pak nachází v adresáři `backend/backend-core/src/api/graphql`.

Mezi přednosti aplikačního rozhraní na bázi technologie GraphQL, oproti běžnějšímu aplikačnímu rozhraní na bázi architektonického stylu REST, patří:

- Efektivní získávání dat – GraphQL umožňuje klientům požadovat přesně ta data, která potřebují, což eliminuje potíže spojené s nadměrným načítáním dat, tzv. *over-fetching*, či naopak nedostatečným načítáním dat, tzv. *under-fetching*. Scénář komunikace mezi klientem a backendem systému je tak omezen na jeden požadavek a jednu odpověď, což je žádoucí z hlediska optimalizace latence, o jejímž významu pojednává podkapitola 3.2. Odpověď navíc neobsahuje nadbytečná data, což je příznivé z hlediska úspory šířky přenosového pásma.
- Výpočty a agregace dat v rámci backendu – GraphQL dokáže zpracovat složité dotazy na data, agregace dat atd. v rámci backendu, čímž se snižuje zátěž na straně klientů.
- Snazší rozvoj aplikačního rozhraní – do aplikačního rozhraní na bázi GraphQL lze přidat další operace a typy, aniž by to ovlivnilo operace a typy, které se využívaly doposud.
- Podpora aktualizací dat v reálném čase – GraphQL umožňuje klientům přihlásit se k odběru aktualizací z backendu pomocí tzv. *subscriptions* na bázi protokolu WebSocket. Oproti technice zvané *API polling*, jejímž principem jsou periodické dotazy ze strany klientů, jde o zlepšení jak z hlediska rychlosti dostupnosti aktualizací, tak zatížení sítě.

Z výše popsaných předností vyplývá, že použití technologie GraphQL pro aplikační rozhraní systému je přínosem z hlediska výkonnosti, efektivity i rozšiřitelnosti systému. Speciální význam měl výše popsaný mechanismus subscriptions, neboť mi umožnil výsledky proudového zpracování dat z chytrých zařízení okamžitě propagovat na klientskou stranu, kde se zobrazí uživateli. Konkrétně se jedná o nejnovější vyhodnocení klíčových indikátorů výkonnosti nad konkrétními zařízeními a záznamy o nových chytrých zařízeních, která systém doposud neevidoval. Propagace těchto aktualizací na klientskou stranu je tak naplněním požadavku, aby systém umožňoval uživatelům, např. personálu chytrých měst, informovaná rozhodnutí založená na nejčerstvějších datech, viz podkapitola 4.2. Pro

implementaci GraphQL serveru v rámci backendu byla použita knihovna gqlgen¹⁴. Pro interakci s aplikačním rozhraním na klientské straně byla použita knihovna Apollo Client¹⁵.

Na určitou komplikaci jsem narazil při snaze o přenos stromových struktur představujících definice klíčových indikátorů výkonnosti. Zatímco u aplikačních rozhraních na bázi architektonického stylu REST klient zašle požadavek na určitý *koncový bod*, z angl. *endpoint*, a dostane data v již zmíněném formátu JSON, který podporuje libovolné zanoření objektů, aplikační rozhraní na bázi technologie GraphQL sice pro transport dat také využívá formát JSON, ale formát GraphQL queries, tedy klientských dotazů, které zároveň určují strukturu odpovědi z backendu, libovolné zanoření nepodporuje. Jednou z možností bylo v queries spojených s načítáním definic klíčových indikátorů výkonnosti definovat určitou úroveň zanoření a spoléhat na to, že bude dostatečná. Lepší možností bylo schéma upravit a stromové struktury přenášet v podobě pole uzlů, přičemž každý uzel disponuje dvěma identifikátory – svým identifikátorem a identifikátorem nadřazeného uzlu. Následně bylo třeba implementovat algoritmus tzv. *zploštění n-árních stromů*, z angl. *n-ary tree flattening*, v rámci backendu a algoritmus pro opětovnou rekonstrukci těchto stromových struktur na klientské straně.

6.3 Služby tvořící backend

Backend systému tvoří, dle návrhu architektury systému popsaného v podkapitole 5.5, trojice služeb: MQTT preprocesor, Message processing unit a Backend core, přičemž každá z těchto služeb zodpovídá za určitý segment celkové funkcionality backendu systému. Služby jsou implementovány jako jedno-modulové aplikace programovacího jazyka Go. Zdrojový kód jednotlivých služeb doplňují externí knihovny, specifikované v souboru `go.mod` modulu dané služby, a také sdílený modul `commons` obsahující zdrojový kód funkcionality užitečné napříč službami – interní knihovnu. Pro připojení sdíleného modulu je použita speciální syntaxe:

```
replace github.com/MichalBures-OG/bp-bures-RIoT-commons => ../../commons
```

Ta specifikuje, že místo URL `github.com/MichalBures-OG/bp-bures-RIoT-commons` má být pro nalezení modulu použita relativní cesta souborového systému: `../../commons`.

6.3.1 Komunikace mezi službami pomocí RabbitMQ

Komunikaci mezi službami zajišťuje technologie RabbitMQ. Základem této technologie je centrální server – broker, resp. jeho open-source implementace, a aplikační protokol AMQP, klíčovým mechanismem pak řazení odeslaných zpráv v rámci *front*, z angl. *queue*, a distribuce zpráv do front pomocí tzv. *exchanges* různého typu. Technologie umožňuje asynchronní zasílání zpráv, flexibilní směrování zpráv a spolehlivé doručení zpráv díky *potvrzením*, z angl. *acknowledgment* a nabízí přehledné uživatelské rozhraní pro správu komunikace. Hodí se pro různě složité komunikační scénáře a poradí si s takřka libovolným objemem zasílaných zpráv – vyniká totiž i z hlediska propustnosti a škálovatelnosti. Mezi její další přednosti patří podpora *prioritizace zpráv*, z angl. *message prioritization*, podpora pro *plánované odeslání zpráv*, z angl. *message scheduling*, a *mechanismy pro automatické opětovné zaslání zpráv*, z angl. *automatic retry mechanisms*.

¹⁴Viz <https://gqlgen.com/>.

¹⁵Viz <https://github.com/apollographql/apollo-client>.

V rámci návrhu architektury systému jsem identifikoval pět scénářů komunikace mezi službami tvořícími backend systému, pro jejichž realizaci bylo třeba zvolit adekvátní prostředky z repertoáru technologie RabbitMQ. Jedná se o následující scénáře:

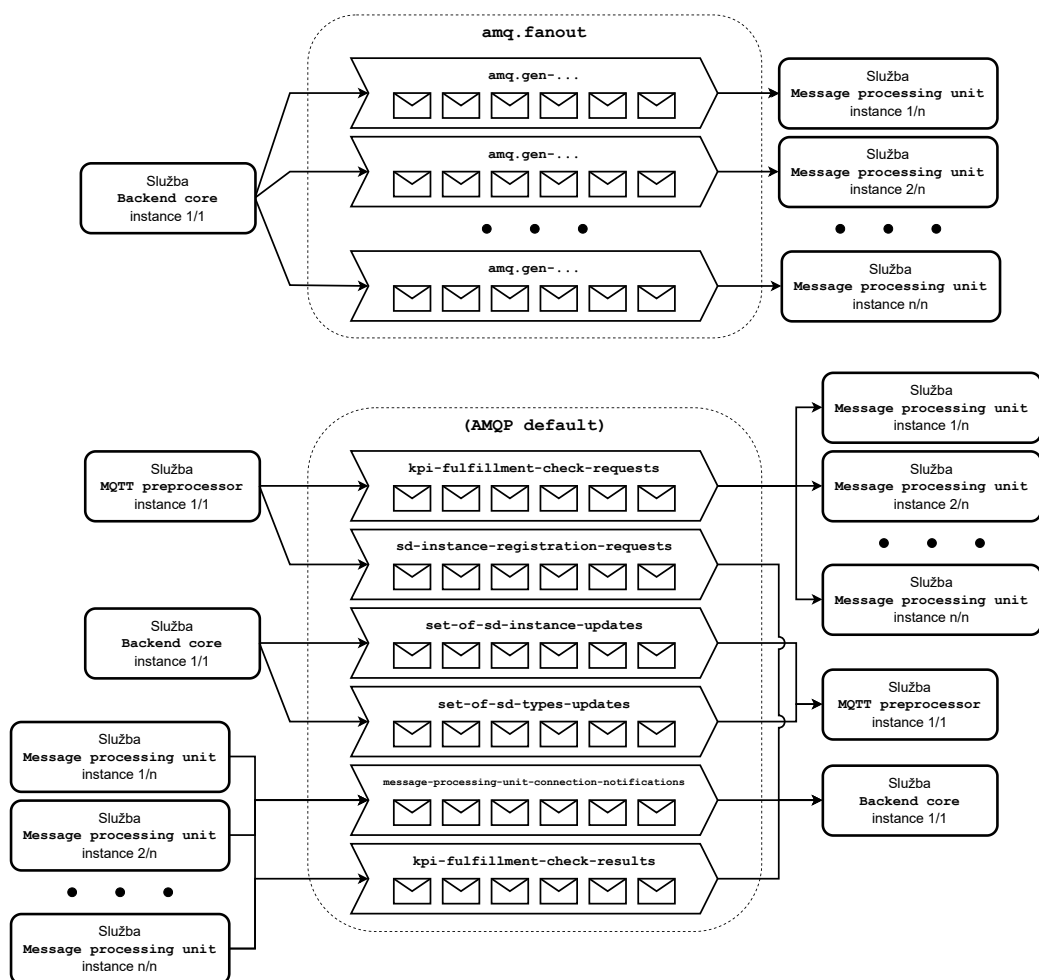
1. Odesílání a příjem žádostí o zaevidování „nových“ zařízení. Odesilatelem je jedna instance služby **MQTT preprocessor** a příjemcem jedna instance služby **Backend core**.
2. Odesílání žádostí o vyhodnocení klíčových indikátorů výkonnosti a jejich rovnoměrné rozdělení mezi příjemce – momentálně běžící instance služby **Message processing unit**. Odesilatelem je jedna instance služby **MQTT preprocessor**.
3. Odesílání a příjem vyhodnocení klíčových indikátorů výkonnosti. Odesilatelé jsou momentálně běžící instance služby **Message processing unit** a příjemcem jedna instance služby **Backend core**.
4. Odesílání a příjem aktualizací množin definic relevantních typů chytrých zařízení a konkrétních chytrých zařízení, která systém již eviduje. Odesilatelem je jedna instance služby **Backend core** a příjemcem jedna instance služby **MQTT preprocessor**.
5. Odesílání aktualizací aktuálně platných definic klíčových indikátorů výkonnosti. Odesilatelem je jedna instance služby **Backend core** a příjemci momentálně běžící instance služby **Message processing unit**.

Pro scénáře 1-4 byly dostačujícím řešením „klasické“ fronty v rámci výchozí exchange typu *direct*, s tím, že jako tzv. *směrovací klíč*, z angl. *routing key*, podle kterého exchange distribuuje zprávy mezi fronty, byl použit identifikátor resp. jméno dané fronty. Scénáře č. 1 a č. 4 jsou triviální, neboť zahrnují pouze jednoho odesilatele a jednoho příjemce – dostatečnost „klasických“ front tak není nikterak překvapivá.

Scénář č. 3 sice zahrnuje předem neznámý počet odesílatelů, ale ti pouze plní danou frontu, takže není nutná žádná další specifická konfigurace. Bez složitějších mechanismů či dodatečné konfigurace se obejde i scénář č. 2, který činí složitějším nutnost co nejrovnoměrnější distribuce příchozích zpráv mezi předem neznámý počet příjemců. I zde postačí „klasická“ fronta – přihlásí-li se k odběru zpráv z fronty vícero příjemců, budou mezi ně rozdělovány podle algoritmu *round-robin*. Principem aplikace tohoto algoritmu je, že první zpráva bude předána prvnímu přihlášenému příjemci, druhá druhému, atp. Poté co bude n zpráv postupně předáno n přihlášeným příjemcům, dojde k 2. iteraci a zpráva s pořadím $n + 1$ tak bude předána prvnímu příjemci, zpráva s pořadím $n + 2$ druhému a tak stále dokola. Tento algoritmus je poměrně triviální a má svá omezení, nicméně za předpokladu spolehlivé činnosti a srovnatelného výkonu příjemců, tedy instancí služby **Message processing unit**, je dostačující.

Jediným scénářem, u kterého „klasická“ fronta v rámci výchozí exchange stačit nebude, je scénář č. 5 – tutěž zprávu je totiž potřeba doručit předem neznámému počtu příjemců, což „klasická“ fronta neumožňuje. V momentě kdy zprávu obdrží některý z příjemců, je z fronty odebrána a není již nadále k dispozici. Zbytek příjemců by se tak ke zprávě nedostal. Bylo by nutné tutěž zprávu odeslat tolikrát, kolik je momentálně přihlášených příjemců, což nejde a i kdyby šlo, nejednalo by se o koncepčně korektní řešení. Tím je využití exchange typu *fanout*. Principem tohoto typu exchange je předání každé z přijatých zpráv do každé z front v rámci této exchange – tento princip se označuje jako *message broadcasting*. Při realizaci komunikačního scénáře č. 5 jsem zajistil, aby každý z příjemců – instancí

služby **Message processing unit**, deklaroval vlastní frontu v rámci výchozí exchange typu fanout. To umožnilo službě **Backend core** zasílat zprávy všem momentálně běžícím instancím služby **Message processing unit**, a to bez ohledu na jejich momentální počet. Určitým úskalím bylo předání konfigurace instancím služby **Message processing unit** po spuštění celého systému – služba **Backend core** je spouštěna jako první a nemůže na exchange typu fanout zasílat zprávy, dokud zde nejsou deklarovány fronty příjemců – došlo by k zahození zpráv. Dále, pokud by se za běhu systému přihlásil nový příjemce – další instance služby **Message processing unit**, nezískal by patřičnou konfiguraci, dokud by na straně služby **Backend core** nevznikl důvod k rozeslání aktualizace. Tento problém jsem vyřešil přidáním další „klasické“ fronty v rámci výchozí exchange typu direct, kam instance služby **Message processing unit** zašlou notifikaci, poté co se připojí, a příjemce – služba **Backend core**, pak skrze exchange typu fanout rozešle aktualizaci, potažmo aktuální konfiguraci. Celkem tak komunikace mezi službami zahrnuje dvě exchanges: výchozí exchange typu direct – (**AMQP default**), a výchozí exchange typu fanout – **amq.fanout**, a $6 + n$ front, přičemž n je počet front, skrze které jsou jednotlivé instance služby **Message processing unit** připojeny k exchange **amq.fanout**. Výše popsanou komunikační architekturu ilustruje obrázek 6.4.



Obrázek 6.4: Diagram komunikace mezi službami pomocí RabbitMQ.

6.3.2 Služba MQTT preprocesor

Implementaci služby MQTT preprocesor obsahuje adresář `mqtt-preprocesor`, který je podadresářem adresáře `backend`. Skládá se z implementace MQTT klienta, opírající se o MQTT klienta Eclipse Paho, kterou obsahuje soubor `mqtt/client.go`, struktury sloužící pro deserializaci těl příchozích MQTT zpráv odpovídajících již zmiňovanému proprietárnímu formátu firmy Logimic – Dev5, která se nachází v souboru `model/logimic.go`, a hlavního zdrojového souboru obsahujícího operační logiku služby – `main.go`. Tato logika řeší příjem a zpracování zpráv ze zadaného tématu¹⁶ MQTT brokeru a to v kontextu aktuální konfigurace systému, jejíž aktualizace přijímá přes RabbitMQ od služby Backend core. Spojení aktuální konfigurace systému pak označuje aktuálně definované relevantní typy zařízení a záznamy o zařízeních, která již systém eviduje, spolu s jejich stavem – již potvrzené či zatím nepotvrzené zařízení. Logika služby MQTT preprocesor pracuje s trojicí globálních proměnných: množinou¹⁷ označení relevantních typů zařízení – `sdTypes`, množinou záznamů o již evidovaných zařízeních – `sdInstances`, a FIFO frontou `mqttMessageFIFO`, do které jsou prostřednictvím callbacku MQTT klienta (`addIncomingMQTTMessageToFIFO`) ukládána těla příchozích zpráv. Výše uvedené proměnné jsem přizpůsobil pro bezpečný souběžný přístup pomocí mutexů¹⁸ – struktura `Mutex` z balíčku `sync`, který je součástí standardní knihovny jazyka. To je potřeba, neboť logika služby zahrnuje trojici goroutines, které k těmto proměnným přistupují:

1. Goroutine, jejíž úlohou je kontrola `mqttMessageFIFO` – obsahuje-li fronta nějaké zprávy, potažmo jejich těla, je třeba je zpracovat, za což zodpovídá funkce `processMQTTMessagePayload`, realizující počáteční segment z diagramu 5.3. Tato funkce tak nejprve deserializuje tělo zprávy, a následně prostřednictvím funkce `mqttMessageSDTypeCorrespondsToSDTypeDefinitions` a obsahu proměnné `sdTypes` zkontroluje typ zařízení, ze kterého zpráva pochází. Pakliže se jedná o typ, který neodpovídá žádné z uživatelských definic relevantních typů zařízení, není třeba se zprávou dále zabývat a funkce `processMQTTMessagePayload` končí. Pakliže jde o relevantní typ zařízení, funkce pokračuje a prostřednictvím funkce `determineSDInstanceScenario` a obsahu proměnné `sdInstances` určí stav daného zařízení: neznámé zařízení, známé zařízení, které zatím nebylo potvrzeno, či známé, potvrzené zařízení, viz podkapitola 5.2. Jde-li o neznámé zařízení, funkce `processMQTTMessagePayload` odešle prostřednictvím funkce `generateSDInstanceRegistrationRequest` žádost o zaevidování daného zařízení – viz komunikační scénář č. 1, popsáný v sekci 6.3.1. Jde-li o známé, potvrzené zařízení, funkce `processMQTTMessagePayload` odešle prostřednictvím funkce `generateKPIFulfillmentCheckRequest` žádost o vyhodnocení klíčových indikátorů výkonnosti nad daty obsaženými ve zprávě – viz komunikační scénář č. 2,

¹⁶Konkrétní téma MQTT brokeru i další konfigurační položky lze zadat pomocí *proměnných prostředí*, angl. *environment variables*. Tento způsob konfigurace se vztahuje jak na backend systému, potažmo trojici služeb, která jej tvoří, tak i frontend systému.

¹⁷Specifikace jazyka Go nezahrnuje vestavěný datový typ množina. Rozhodl jsem se proto tento typ implementovat sám v rámci sdíleného modulu `commons` – soubor `src/sharedUtils/set.go`, a využil jsem přitom vestavěný datový typ `map`, viz <https://go.dev/blog/maps>.

¹⁸Jako mutex, z angl. slovního spojení *mutual exclusion*, česky *vzájemné vyloučení*, se označuje synchronizační primitivum, pomocí kterého lze zabránit nežádoucímu současnému přístupu ke sdílenému zdroji, či kritické sekci programu.

popsaný v sekci 6.3.1. Jde-li o známé zařízení, které zatím nebylo potvrzeno, funkce `processMQTTMessagePayload` končí.

2. Goroutine která zodpovídá za aktualizace definic relevantních typů chytrých zařízení – prostřednictvím funkce `checkForSetOfSDTypesUpdates` odebírá zprávy z RabbitMQ fronty `set-of-sd-types-updates` a na základě těchto zpráv mění hodnotu proměnné `sdTypes`.
3. Goroutine která zodpovídá za aktualizace záznamů o zařízeních, která již systém eviduje, a jejich stavu – prostřednictvím funkce `checkForSetOfSDInstancesUpdates` odebírá zprávy z RabbitMQ fronty `set-of-sd-instances-updates` a na základě těchto zpráv mění hodnotu proměnné `sdInstances`.

V jednoduchosti lze říci, že služba MQTT `preprocessor` transformuje MQTT komunikaci na interní systémové zprávy dvojího typu: žádosti o za evidování „nových“ zařízení a žádosti o vyhodnocení klíčových identifikátorů výkonnosti, a zohledňuje přitom uživatelskou konfiguraci relevantních typů chytrých zařízení. Pokud by měl systém přijímat zprávy z chytrých zařízení i z dalších zdrojů, mimo MQTT komunikaci, stačilo by tuto službu rozšířit a přejmenovat, a nebo přidat službu, která by byla analogií této služby.

6.3.3 Služba Message processing unit

Implementaci služby, nacházející se v adresáři `message-processing-unit`, který je podadresářem adresáře `backend`, tvoří pouze dva zdrojové soubory – soubor `src/processing/kpiFulfillmentCheck.go` obsahující kód realizující samotné vyhodnocení stromových struktur představujících definice klíčových indikátorů výkonnosti a soubor `src/main.go` obsahující kód zbytku operační logiky služby. Implementace služby zahrnuje pouze jednu globální proměnnou s označením `kpiDefinitionsBySDTypeDenotationMap`, přičemž bezpečný souběžný přístup k této proměnné je opět řešen pomocí mutexu. Výše zmíněná proměnná je typu `map`, tedy *hashovací tabulka*, z angl. *hash table*. Klíči tabulky jsou označení typů chytrých zařízení a hodnotami množiny definic klíčových indikátorů výkonnosti definovaných pro daný typ chytrého zařízení. Operační logika služby zahrnuje dvojici goroutines, které k této proměnné přistupují:

1. Goroutine, která na základě žádostí zasílaných službou MQTT `preprocessor` provádí vyhodnocení klíčových indikátorů výkonnosti. Žádosti, se kterými tato goroutine pracuje, představují zprávy odebírané z RabbitMQ fronty `kpi-fulfillment-check-requests`, které obsahují UID konkrétního zařízení chytrého zařízení, označení typu tohoto zařízení a parametry, dané typem zařízení, s hodnotami pocházejícími z daného chytrého zařízení. Goroutine tak použije označení typu chytrého zařízení jako klíč na jehož základě získá z proměnné `kpiDefinitionsBySDTypeDenotationMap` aktuálně platné definice klíčových indikátorů výkonnosti definované pro daný typ chytrého zařízení, které skrze funkci `CheckKPIFulfillment` vyhodnotí na základě hodnot parametrů zařízení obsažených v dané žádosti. Vlastnímu vyhodnocení předchází kontrola, jestli je daná definice platná pro dané konkrétní zařízení. To platí, pokud je definice platná pro všechna zařízení daného typu:

```
kpiDefinition.SDInstanceMode == sharedModel.ALL
```

a nebo pouze pro některá, přičemž množina unikátních identifikátorů zařízení, pro která je definice platná, zahrnuje unikátní identifikátor daného zařízení:

```
selectedSDInstanceUIDSet.Contains(sdInstanceUID)
```

Vyhodnocení n definic platných pro dané zařízení probíhá paralelně, pomocí n goroutines, a je zakončeno odesláním zprávy – výsledku vyhodnocení do RabbitMQ fronty `kpi-fulfillment-check-results`.

2. Goroutine která zodpovídá za aktualizace obsahu hashovací tabulky, tedy proměnné `kpiDefinitionsBySDTypeDenotationMap`, na základě zpráv, které služba **Backend core** rozesílá jednotlivým instancím služby **Message processing unit** pomocí exchange typu `fanout` a které tyto instance odebírají pomocí svých vlastních front – `amq.gen-...`, viz obrázek 6.4.

6.3.4 Služba Backend core

Oproti předchozím službám plní služba **Backend core** podstatně více úloh a její implementace, obsažená v adresáři `backend-core`, který je podadresářem adresáře `backend`, je tak nejrozsáhlejší. Je proto rozdělená mezi mnoho tzv. *balíčků*, z angl. *packages*, které definuje jednak adresář souborového systému a také deklarace, která se umísťuje na začátek zdrojových souborů, např.:

```
package graphql
```

Předností organizace kódu do těchto balíčků je především omezení viditelnosti určitých proměnných a funkcí mimo balíček, což zvyšuje přehlednost základny zdrojového kódu. Každý z balíčků by měl sdružovat kód zodpovědný za určitou oblast funkcionality služby. Rozdělení funkcionality služby **Backend core** mezi balíčky popisuje tabulka, která byla pro přehlednost rozdělena na dvě části: tabulka 6.1 a tabulka 6.2.

Označení balíčku	Funkcionalita
<code>graphql</code>	Implementace aplikačního rozhraní služby na bázi technologie GraphQL: schéma a kód resolver functions.
<code>gsc</code>	Implementace aplikačního rozhraní služby na bázi technologie GraphQL: doplňující, obslužný kód vygenerovaný skrze knihovnu <code>gqlgen</code> .
<code>dbClient</code>	Implementace klienta relační databáze PostgreSQL pomocí ORM knihovny GORM ¹⁹ : samotný klient.
<code>dbUtil</code>	Implementace klienta relační databáze PostgreSQL pomocí ORM knihovny GORM: pomocné funkce týkající se obecných databázových operací.
<code>domainLogicLayer</code>	Doménová logika služby, resp. celého systému.

Tabulka 6.1: Rozdělení funkcionality služby **Backend core** mezi balíčky – 1. část

¹⁹Viz <https://gorm.io/index.html>.

Označení balíčku	Funkcionalita
<code>isc</code>	Komunikace mezi službou Backend core a ostatními službami tvořícími backend systému – zasílání a odběr RabbitMQ zpráv.
<code>dbModel</code>	Datový model systému na úrovni ORM knihovny.
<code>d11Model</code>	Datový model systému na úrovni doménové logiky.
<code>graphqlModel</code>	Datový model systému na úrovni aplikačního rozhraní. Kód představující tento model byl vygenerován pomocí knihovny <code>gqlgen</code> na základě schématu aplikačního rozhraní – soubor <code>schema.graphqls</code> .
<code>db2d11</code>	Transformace prvků datového modelu systému na úrovni ORM knihovny na korespondující prvky datového modelu na úrovni doménové logiky.
<code>d112db</code>	Transformace prvků datového modelu systému na úrovni doménové logiky na korespondující prvky datového modelu na úrovni ORM knihovny.
<code>d112gql</code>	Transformace prvků datového modelu systému na úrovni doménové logiky na korespondující prvky datového modelu na úrovni aplikačního rozhraní.
<code>gql2d11</code>	Transformace prvků datového modelu systému na úrovni aplikačního rozhraní na korespondující prvky datového modelu na úrovni doménové logiky.

Tabulka 6.2: Rozdělení funkcionality služby **Backend core** mezi balíčky – 2. část

Tato služba je jedinou službou, se kterou komunikuje klientská strana neboli frontend systému, a funguje tak jako gateway celého backendu. Typický scénář interakce mezi frontendem a backendem, potažmo službou **Backend core**, zahrnuje tyto kroky:

1. Uživatel zadá požadavek skrze uživatelské rozhraní systému, které je součástí frontendu.
2. Požadavek dorazí v podobě konkrétní query či mutace na aplikační rozhraní služby.
3. Proveďte se kód příslušné resolver function ze souboru `schema.resolvers.go`, který je součástí balíčku `graphql`.
4. Z kódu resolver function se zavolá příslušná funkce doménové logiky z balíčku `domainLogicLayer`.
5. Tato funkce skrze klienta relační databáze – soubor `client.go`, který je součástí balíčku `dbClient`, provede patřičné databázové operace.
6. Dalším krokem, ke kterému může a nemusí dojít, je zaslání patřičných notifikací o změně konfigurace systému službě MQTT `preprocesor` a instancím služby `Message processing unit` skrze RabbitMQ. To je z praktického hlediska zajištěno zavoláním odpovídající funkce z balíčku `isc`.
7. Posledním krokem scénáře je propagace výsledku operace na frontend skrze aplikační rozhraní. To je z praktického hlediska zajištěno pomocí návratových hodnot funkce

doménové logiky a resolver function, přičemž detaily komunikace po síti jsou záležitostí knihovny gqlgen, potažmo její konfigurace – soubor `gqlgen.yml`.

Další úlohou, kterou služba plní, je příjem žádostí o zaevidování „nových“ chytrých zařízení, které jí zasílá služba `MQTT preprocessor`, a výsledků vyhodnocení klíčových indikátorů výkonnosti, které jí zasílají instance služby `Message processing unit`. V obou případech jde o zprávy odebírané z front technologie RabbitMQ, jejichž obsah služba `Backend core` nejprve uloží do relační databáze²⁰ a následně propaguje na frontend pomocí GraphQL subscription.

6.3.5 Sdílený modul `commons`

Sdílený modul `commons` obsahuje zdrojový kód, který využívají alespoň dvě z výše popsanych služeb. Je rozdělen na čtyři části, resp. balíčky, jejichž účel popisuje tabulka 6.3.

Označení balíčku	Účel balíčku
<code>rabbitmq</code>	Implementace RabbitMQ klienta, používaného napříč službami.
<code>sharedConstants</code>	Konstanty používané napříč službami – především identifikátory spojené s technologií RabbitMQ.
<code>sharedModel</code>	Část datového modelu systému sdílená napříč službami: sdílený model definic klíčových indikátorů výkonnosti – soubor <code>kpiDefinitionModel.go</code> , a model RabbitMQ zpráv – soubor <code>iscMessages.go</code> .
<code>sharedUtils</code>	Kolekce různorodých pomocných funkcí a datových struktur používaných napříč službami.

Tabulka 6.3: Balíčky sdíleného modulu `commons` a jejich účel.

Speciální pozornost si zaslouží zejména obsah balíčku `sharedUtils`. Jednotlivé zdrojové soubory tvořící obsah tohoto balíčku a jejich význam popisuje tabulka, která byla pro přehlednost rozdělena na dvě části: tabulka 6.4 a tabulka 6.5.

Zdrojový soubor	Význam zdrojového souboru
<code>concurrentCounter.go</code>	Datová struktura <code>ConcurrentCounter</code> představující čítač umožňující souběžný přístup z několika goroutines.
<code>debug.go</code>	Snazší ladění pomocí detailního výpisu obsahu proměnných pomocí funkce <code>Dump</code> , přičemž tato funkce se opírá o stejnojmennou funkci z knihovny <code>github.com/davecgh/go-spew/spew</code> .
<code>dsReadiness.go</code>	Kontrola připravenosti služeb běžících jako kontejnery technologie Docker, na kterých daná služba závisí.

Tabulka 6.4: Zdrojové soubory z balíčku `sharedUtils` a jejich význam – 1. část

²⁰Náročnost s tímto spojených databázových operací byla snížena zavedením patřičných indexů, jejichž definice jsou součástí obsahu souboru `model.go` z balíčku `dbModel`.

Zdrojový soubor	Význam zdrojového souboru
<code>environment.go</code>	Načítání hodnot z proměnných prostředí, použitých pro konfiguraci služeb tvořících backend systému, skrze funkci <code>GetEnvironmentVariableValue</code> .
<code>errorHandling.go</code>	Pomocné funkce pro jednodušší ošetření chyb programu.
<code>fpHelpers.go</code>	Pomocné funkce zjednodušující práci s proměnnými typu <code>slice</code> , imitující určité aspekty funkcionálního programování.
<code>generators.go</code>	Generátor číselné posloupnosti pro datový typ <code>uint32</code> , umožňující souběžný přístup z několika goroutines.
<code>json.go</code>	Pomocné funkce usnadňující serializaci proměnných do formátu JSON a také deserializaci dat v tomto formátu. Implementace těchto funkcí se opírá o balíček standardní knihovny jazyka Go – <code>encoding/json</code> .
<code>optional.go</code>	Datová struktura <code>Optional[T any]</code> pro reprezentaci hodnot různého typu, které mohou potenciálně chybět, nad rámec ukazatelů a speciální hodnoty <code>nil</code> .
<code>pair.go</code>	Datová struktura <code>Pair[T any, U any]</code> pro reprezentaci dvojice hodnot, potenciálně různého typu, mezi kterými je úzká souvislost.
<code>profiling.go</code>	Pomocná funkce pro periodický výpis informací užitečných při profilování programu, např. objem alokované operační paměti.
<code>result.go</code>	Datová struktura <code>Result[T any]</code> pro reprezentaci výsledku operace, která mohla potenciálně selhat. Tato struktura se používá napříč implementací všech služeb tvořících backend systému.
<code>set.go</code>	Datový typ množina, jehož implementace proběhla pomocí vestavěného datového typu <code>map</code> .
<code>slice.go</code>	Pomocné funkce pro snazší definici proměnných typu <code>slice</code> .
<code>sync.go</code>	Pomocná funkce <code>WaitForAll</code> umožňující současné spuštění a paralelní běh několika funkcí pomocí goroutines, s tím, že primární goroutine „čeká“, dokud nedoběhne poslední z těchto funkcí.
<code>ternaryOperator.go</code>	Pomocná funkce <code>Ternary[T any]</code> , která nahrazuje tzv. ternární operátor, který standardní syntaxe jazyka Go neobsahuje.
<code>typeSystem.go</code>	Pomocná funkce <code>TypeIs[T any]</code> která usnadňuje kontrolu, zdali typ konkrétní proměnné odpovídá zadanému typu.

Tabulka 6.5: Zdrojové soubory z balíčku `sharedUtils` a jejich význam – 2. část

Výše zmíněné pomocné funkce a datové struktury zdrojový kód zpřehledňují, a činí jej méně náchylným k chybám a to díky abstrakcím a centralizaci implementace často používaných operací. Důležitou informací je to, že implementace většiny z těchto pomocných funkcí a

datových struktur by nebyla možná bez tzv. *generik*, z angl. *generics*, která byla do jazyka Go přidána ve verzi 1.18 a která umožňují implementaci datových struktur a algoritmů nezávislých na konkrétním datovém typu či rozhraní.

Kapitola 7

Testování systému

Při testování systému pro zpracování dat z chytrých zařízení šlo o 2 primární hlediska: testování spolehlivosti, potažmo funkčnosti systému, a testování výkonnosti systému. První hledisko jsem chtěl ošetřit skrze rozsáhlou kolekci tzv. *jednotkových testů*, z angl. *unit tests*. To se mi nicméně s ohledem na časovou náročnost implementace těchto jednotkových testů nepodařilo. Implementoval jsem pouze dva a to jako příklady, užitečné pro případný budoucí rozvoj systému zahrnující implementaci mnoha dalších jednotkových testů.

- První jednotkový test, implementovaný pomocí knihovny `go-sqlmock`¹, ověřuje bezchybnost konkrétní operace klienta relační databáze – uložení definice relevantního typu chytrého zařízení, a je obsahem souboru `client_test.go`², z balíčku `dbClient` služby `Backend core`.
- Druhý jednotkový test, implementovaný pomocí knihovny `go-cmp`³, jenž je obsahem souboru `sdTypes_test.go` z balíčku `db2dll` téže služby, ověřuje bezchybnost transformace datového modelu definic relevantních typů chytrých zařízení na úrovni ORM knihovny na korespondující model na úrovni doménové logiky.

Systém lze pomocí implementovaných jednotkových testů otestovat pomocí příkazu `go test ./...`, přičemž specifikace cíle `./...` značí veškeré jednotkové testy v rámci daného adresáře – ideálně pak kořenového adresáře implementace vybrané služby. U služby `Backend core` je tento příkaz součástí `Dockerfile`, tedy souboru pro konstrukci tzv. *Docker image* – „šablony“ pro kontejner služby. To zajišťuje, že funkcionality služby je pomocí implementovaných jednotkových testů testována průběžně, pokaždé když dochází ke konstrukci `Docker image` služby, což je nezbytné, aby se projevil efekt modifikace zdrojového kódu služby. Dalším logickým krokem je tedy nastavení výše popsaného mechanismu u zbylých dvou služeb a především implementace mnoha dalších jednotkových testů. Za nejdůležitější považuji implementaci jednotkových testů týkajících se databázových operací a transformací prvků datového modelu napříč úrovněmi, viz tabulka 6.2. Korektní funkcionality backendu systému jsem tak, s ohledem na minimální množství jednotkových

¹Viz <https://github.com/DATA-DOG/go-sqlmock>.

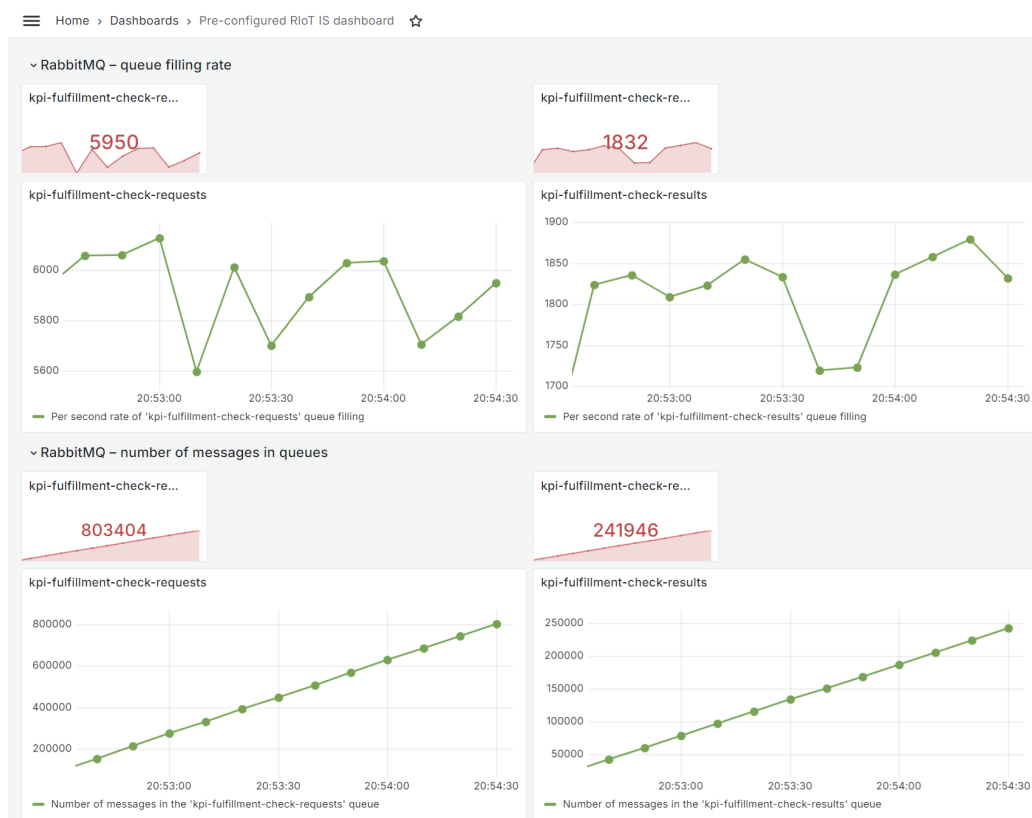
²Suffix `_test` má speciální význam. Značí, že daný zdrojový soubor obsahuje samostatně spustitelné testy, typicky jednotkové. Tento suffix je užitečný jak pro příkaz `go test`, tak pro nástroje, které jsou součástí integrovaného vývojového prostředí, *IDE* z angl. *Integrated Development Environment*.

³Viz <https://github.com/google/go-cmp>.

testů, nakonec testoval především manuálně, s použitím nástrojů Apollo Sandbox⁴, pgAdmin⁵ atd. Korektní funkcionality frontendu systému jsem ověřoval taktéž manuálně skrze simulovanou uživatelskou interakci. I zde jsem při testování a ladění využil různé nástroje, především pak sadu vývojářských nástrojů zabudovaných do webového prohlížeče označovanou jako DevTools⁶.

7.1 Měření výkonnosti systému

Po ověření funkčnosti systému bylo možné začít měřit a vyhodnocovat jeho výkonnost. Jako hlavní kritéria, jsem vzhledem k povaze systému zvolil propustnost a *celosystémovou latenci*, z angl. *end-to-end latency*, tedy jaký objem zpráv z chytrých zařízení systém dokáže zpracovat za jednotku času – 1 sekundu, a kolik času zabere, než zpráva „projde systémem“ a promítne se do uživatelského rozhraní. Při měření propustnosti systému jsem použil již zmiňovanou populární kombinaci nástrojů – Prometheus a Grafana, a zkonstruoval pomocí těchto nástrojů dashboard, který seskupuje podstatné metriky a poskytuje tak dokonalejší přehled o interních záležitostech systému.



Obrázek 7.1: Grafana – dashboard

⁴Nástroj umožňující introspekci a ladění aplikačních rozhraní na bázi technologie GraphQL, který je dostupný skrze uživatelské rozhraní systému. Oficiální dokumentace viz <https://www.apollographql.com/docs/graphos/explorer/sandbox/>.

⁵Platforma pro administraci a ladění relační databáze PostgreSQL. Více viz <https://www.pgadmin.org/>.

⁶Viz <https://developer.chrome.com/docs/devtools>.

S ohledem na jeho potenciální užitečnost nad rámec měření propustnosti systému jsem soubor, obsahující definici tohoto dashboardu, zahrnul do repositáře projektu a zařídil, že se do nástroje Grafana, který běží jako samostatný kontejner, naimportuje automaticky. Obsahem již zmíněného dashboardu, který zobrazuje obrázek 7.1 je deset metrik, z čehož devět se týká technologie RabbitMQ. Jde o množství zpráv ve třech klíčových frontách, tedy frontě požadavků na vyhodnocení klíčových indikátorů výkonnosti – `kpi-fulfillment-check-requests`, frontě výsledků vyhodnocení klíčových indikátorů výkonnosti – `kpi-fulfillment-check-results`, a frontě požadavků na zaevidování „nového“ zařízení – `sd-instance-registration-requests`, rychlost plnění těchto front a počet otevřených spojení a kanálů⁷. 10. metrikou je frekvence s jakou na dané téma MQTT brokeru přicházejí zprávy. To je podstatné z hlediska propustnosti služby MQTT `preprocessor`, která na základě těchto příchozích zpráv generuje systémové zprávy, kterými plní fronty a `kpi-fulfillment-check-requests` – frontu požadavků na vyhodnocení klíčových indikátorů výkonnosti, a `sd-instance-registration-requests` – frontu požadavků na zaevidování „nového“ zařízení. Její propustnost jsem měřil na základě umělého proudu MQTT zpráv s předem danými charakteristikami, přičemž jsem odpojl instance služby `Message processing unit` a následně pozoroval, jestli rychlost plnění fronty odpovídá objemu příchozích MQTT zpráv, které jsem generoval pomocí skriptu `mqtt_stress_generator.py`, jenž je též součástí repositáře projektu.

7.1.1 Odhad propustnosti služby MQTT `preprocessor`

Před měřením reálné propustnosti služby pomocí nástroje Grafana jsem se rozhodl propustnost služby odhadnout na základě času, který zabere zpracování jedné zprávy. Ten jsem měřil pomocí dvojice tzv. *sond*⁸, z angl. *probes*, které jsem umístil do kódu služby – na začátek a na konec těla funkce `processMQTTMessagePayload`. Ukázalo se, že zpracování jedné zprávy touto službou, počínaje deserializací zprávy a konče odesláním požadavku na vyhodnocení klíčových indikátorů výkonnosti, zabere nižší desetiny milisekundy. Průměr ze 100 takto naměřených hodnot byl $348 \mu s$ ⁹, tedy přibližně 0.35 ms. Odhad propustnosti služby MQTT `preprocessor` tak činil necelých 3000 zpráv za sekundu.

7.1.2 Reálná propustnost služby MQTT `preprocessor`

Reálná propustnost služby, kterou ilustruje graf na obrázku 7.2, pak byla ještě vyšší. Tento graf ukazuje rychlost plnění fronty `kpi-fulfillment-check-requests`, která se při splnění následujících předpokladů rovná propustnosti.:

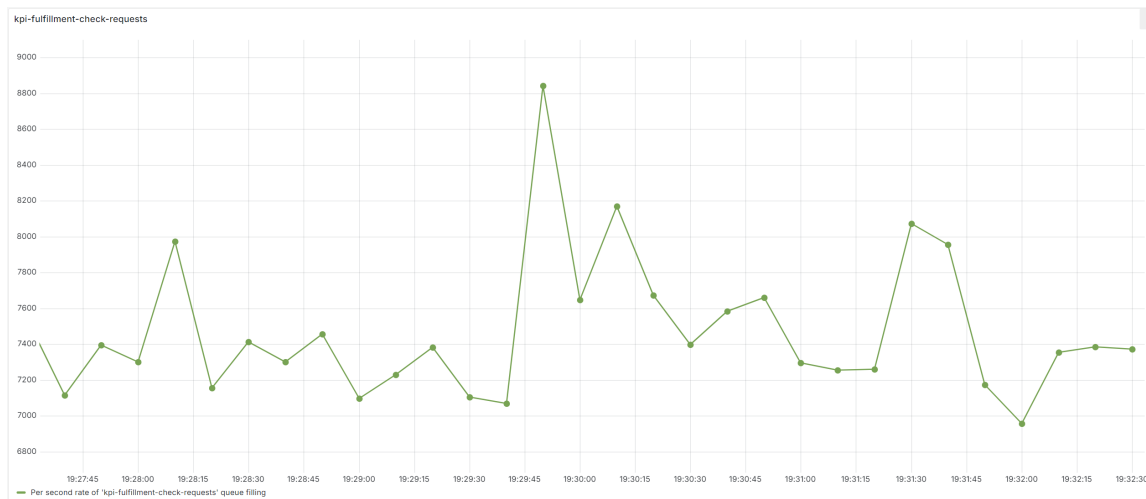
- odpojení instancí služby `Message processing unit`,
- generování proudu MQTT zpráv spojených se zařízeními, jejichž typ odpovídá některé z definic relevantních typů zařízení a která již byla potvrzena uživatelem.

⁷Zatímco *spojení*, z angl. *connection*, je skutečné TCP spojení, *kanál*, z angl. *channel*, je odlehčené, virtuální spojení, které může být otevřeno i uzavřeno s minimální režii. Primárním principem je zde sdílení jednoho TCP spojení vícero vláken či procesy.

⁸Pro získání aktuálního času s přesností na mikrosekundy jsem použil příkaz `time.Now().UnixMicro()`.

⁹Tato hodnota byla společně s dalšími naměřena s použitím mého notebooku, disponujícího procesorem Intel® Core™ i7-12700H a 32 GB operační paměti typu DDR5 SDRAM s taktom 4800 MHz, na kterém byl systém spuštěn skrze nástroj Docker Compose. Na téže stroji současně běžel i již zmíněný skript pro generování zátěže.

Z grafu lze vyčíst, že reálná propustnost služby **MQTT preprocesor** určená výše popsanou metodou činila přibližně 7800 zpráv za sekundu. To koresponduje s maximální zátěží, kterou jsem byl schopen vygenerovat – limitujícími faktory byly použitý Python skript i MQTT broker běžící jako Docker kontejner.



Obrázek 7.2: Rychlost plnění fronty `kpi-fulfillment-check-requests`

7.1.3 Propustnost služby **Message processing unit**

Propustnost této služby, tedy množství požadavků na vyhodnocení klíčových indikátorů výkonnosti, které je služba schopná obsloužit za sekundu, závisí na dvou faktorech: počtu definic klíčových indikátorů výkonnosti, které je třeba na základě každého požadavku vyhodnotit, a jejich složitosti. Zvolil jsem scénář zahrnující vyhodnocení tří středně složitých definic, tzn. definic, jejichž modelem byl strom s hloubkou 3 u kterého 1. úroveň tvořila jedna logická operace, 2. úroveň dvě logické operace a 3. úroveň šest „atomů“¹⁰. Stejně jako u služby **MQTT preprocesor** jsem i zde začal odhadem propustnosti pomocí dvojice sond, které jsem tentokrát umístil do funkce `checkForkKPIFulfillmentCheckRequests`. Na základě průměru ze 100 měření, který činil 2818 μ s, tedy přibližně 2,8 ms, jsem propustnost služby odhadl jako přibližně 360 zpráv, resp. požadavků na vyhodnocení klíčových indikátorů výkonnosti, za sekundu. Reálnou propustnost služby **Message processing unit** jsem následně určil níže popsanou metodou.:

1. Odpojíl jsem běžící instance služby **Message processing unit**.
2. Pomocí již zmiňovaného Python skriptu jsem vygeneroval značné množství MQTT zpráv a na jejichž základě služba **MQTT preprocesor** naplnila frontu požadavků na vyhodnocení klíčových indikátorů výkonnosti – `kpi-fulfillment-check-requests`.
3. Odpojíl jsem skript generující MQTT zprávy, což mělo za následek ustálení množství zpráv ve výše uvedené frontě.
4. Zapojil jsem jednu instanci služby **Message processing unit** a pomocí nástroje Grafana sledoval pokles množství zpráv v dané frontě – ten zachycuje graf na obrázku [7.3](#).

¹⁰Takto koncipovanou definici si lze prohlédnout na obrázku [5.2](#).



Obrázek 7.3: Pokles množství zpráv ve frontě `kpi-fulfillment-check-requests`

Z grafu na obrázku 7.3 lze vyčíst, že za 290 sekund se množství zpráv ve frontě `kpi-fulfillment-check-requests` snížilo z 318518 na 152666. Výslednou, reálnou propustnost služby jsem tak získal jednoduchým výpočtem:

$$\frac{318518 - 152666}{290} = \frac{165852}{290} \approx 572$$

Výsledek – přibližně 572 zpráv, potažmo požadavků na vyhodnocení klíčových indikátorů výkonnosti, za sekundu, je věrohodný. Z grafu na obrázku 7.3 je patrné, že propustnost služby byla poměrně stabilní a do měření propustnosti zde nezasahovalo ani generování zátěže, jako tomu bylo při měření propustnosti služby MQTT preprocessor.

7.1.4 Propustnost služby Backend core

Z hlediska propustnosti služby Backend core, která disponuje nejrozsáhlejším portfoliem úkonů, jsem se zaměřil na příjem výsledků vyhodnocení klíčových indikátorů výkonnosti, jejich uložení do relační databáze a jejich propagaci na klientskou stranu systému, přičemž za tuto funkcionalitu zodpovídá funkce `ProcessIncomingKPIFulfillmentCheckResults`. Postup určení propustnosti služby byl obdobou postupu uplatněného u služby `Message processing unit` – začal jsem odhadem propustnosti služby na základě měření času potřebného pro zpracování jedné zprávy s použitím dvojice sond, které jsem umístil do funkce `ProcessIncomingKPIFulfillmentCheckResults`, a následně měřil reálnou propustnost služby na základě poklesu množství zpráv ve frontě výsledků vyhodnocení klíčových indikátorů výkonnosti – `kpi-fulfillment-check-results`. Průměrný čas potřebný pro zpracování jedné zprávy, resp. výsledku vyhodnocení konkrétního klíčového indikátoru výkonnosti, činil 5716 μ s, tedy necelých 6 ms. Odhad propustnosti služby tak činil 175 zpráv, potažmo výsledků vyhodnocení klíčových indikátorů výkonnosti, za sekundu. Reálná propustnost služby, měřená pomocí nástroje Grafana, pak byla opět o něco vyšší – 286 zpráv, potažmo výsledků vyhodnocení klíčových indikátorů výkonnosti, za sekundu.

7.1.5 Propustnost celého systému

Je zřejmé, že propustnost systému skládajícího se z několika komponent s různou propustností bude omezena tou nejnižší. Tou byla propustnost služby **Backend core**, přičemž „zaručenou“ propustnost této služby lze, s přihlédnutím k potenciálním dopadům ad-hoc CRUD operací, které nebyly do předchozího měření zahrnuty, jen těžko odhadnout. Důležitou roli z hlediska propustnosti systému hraje to, na základě jakého podílu z příchozích zpráv z chytrých zařízení proběhne vyhodnocení klíčových indikátorů výkonnosti a kolik bude vyhodnocováno definic klíčových indikátorů výkonnosti. V rámci výše popsaného měření jsem pracoval se scénářem, u kterého tento podíl činil 100% a u kterého na základě každé zprávy z chytrého zařízení docházelo k vyhodnocení tří definic klíčových indikátorů výkonnosti, přičemž reálná situace může být diametrálně odlišná.

7.2 Optimalizace propustnosti systému

Při měření propustnosti jednotlivých služeb tvořících backend systému byly patrné značné rozdíly a prostor pro zlepšení, potažmo optimalizaci. Cílem optimalizace bylo ošetřit následující nedostatky:

1. Zbytečně vysoká úroveň paralelismu při vyhodnocování definic klíčových indikátorů výkonnosti.
2. Odesílání každého z n -tice výsledků vyhodnocení klíčových indikátorů výkonnosti, které vznikly na základě téhož požadavku, samostatně.
3. Provádění trojice „pomalých“¹¹ databázových operací při zpracování každého výsledku vyhodnocení klíčového indikátoru výkonnosti.

První nedostatek z výše uvedeného seznamu se týkal služby **Message processing unit**. Podstatou tohoto nedostatku bylo, že pro vyhodnocení každé jedné definice klíčového indikátoru výkonnosti byla vytvořena a následně použita samostatná goroutine, přičemž pro odeslání výsledku vyhodnocení byl vytvořen a následně použit samostatný RabbitMQ kanál. Ukázalo se, že režie spojená s tvorbou samostatné goroutine a samostatného RabbitMQ kanálu převyšuje přínos paralelního vyhodnocení středně složitých definic klíčových indikátorů výkonnosti. Tento kontraproduktivní paralelismus jsem tak nahradil sekvenčním zpracováním a následně opět změřil reálnou propustnost služby **Message processing unit** stejnou metodou, jako předtím – tu popisuje sekce 7.1.3. Výsledná propustnost služby po optimalizaci činila 2654 zpráv, resp. požadavků na vyhodnocení klíčových indikátorů výkonnosti, za sekundu. To znamená, že na základě výše popsané optimalizace došlo k nárůstu propustnosti o 364%. I přes tento přesvědčivý výsledek je třeba podotknout, že s nyní aplikovaným sekvenčním zpracováním souvisí jasná omezení – pokud bude třeba na základě každé MQTT zprávy vyhodnocovat výrazně vyšší množství definic, nebo velmi složité definice, bude určitá úroveň paralelismu opět žádoucí. Jako ideální prostředek se zde jeví tzv. *worker pool*, jehož principem je rozdělování úloh mezi omezený počet vláken či goroutines. Na jeho implementaci společně s určením optimálního počtu goroutines jsem již bohužel neměl prostor.

Druhý a třetí nedostatek měly význam z hlediska propustnosti služby **Backend core**, která byla nejnižší a představovala tak tzv. úzké hrdlo, z angl. *bottleneck*, systému. Příčinou

¹¹ Pomalost databázových operací je relativní – zdají se pomalé oproti zbytku operací, které systém provádí.

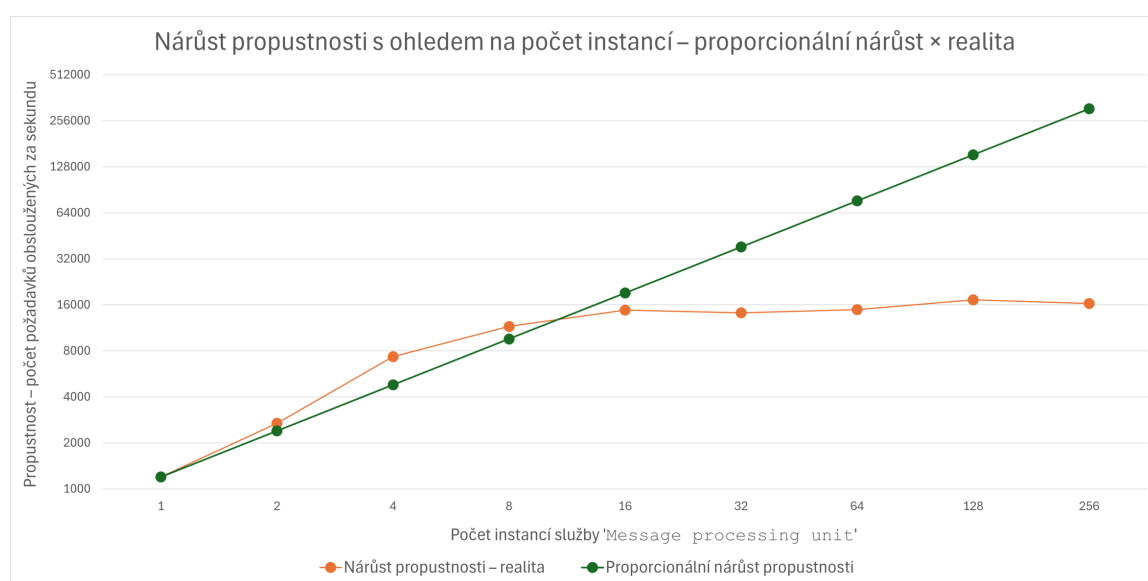
neutěšené situace ohledně omezené propustnosti služby **Backend core** bylo, že služba, s ohledem na integritu ukládaných dat před uložením přijatého výsledku vyhodnocení klíčového indikátoru výkonnosti, kontrolovala existenci definice daného klíčového indikátoru výkonnosti a také existenci záznamu o daném chytrém zařízení, což byly hned dvě „drahé“ databázové operace. Třetí databázovou operací pak bylo samotné uložení výsledku. Tato trojice operací probíhala nad každým přijatým výsledkem, včetně výsledků, které vznikly na základě téhož požadavku – služba **Message processing unit** je totiž odesílala samostatně, jeden po druhém. Principem optimalizace tak bylo sdružení výsledků, které vznikly na základě téhož požadavku, který sám vznikl na základě MQTT zprávy z téhož zařízení, a jejich společné odeslání (službou **Message processing unit**), společné uložení (službou **Backend core**) a společná propagace na klientskou stranu (opět službou **Backend core**). Tím došlo k redukci počtu databázových operací potřebných k uložení n -tice výsledků vyhodnocení klíčových indikátorů výkonnosti z $3 \times n$ (dva dotazy a jeden zápis $\times$ počet prvků n -tice) na $2 + n$ (dva dotazy, bez ohledu na počet prvků n -tice, a jeden zápis $\times$ počet prvků n -tice). Po provedení této optimalizace jsem opět změřil propustnost služby **Backend core** stejným způsobem jako předtím (tento způsob popisuje sekce 7.1.4). Výsledná propustnost služby po optimalizaci činila 173 zpráv za sekundu, což je na první pohled méně, než před optimalizací naměřených 286 zpráv za sekundu. Klíčem k pochopení přínosu optimalizace byl rozdílný význam pojmu zpráva. Zatímco před optimalizací šlo o výsledek vyhodnocení konkrétního klíčového indikátoru výkonnosti, po optimalizaci šlo o celou n -tici těchto výsledků, vzniklou na základě jedné MQTT zprávy z chytrého zařízení. Za účelem přesného zhodnocení přínosu optimalizace jsem tak propustnost služby po optimalizaci vynásobil koeficientem odpovídajícím počtu definic klíčových indikátorů výkonnosti, k jejichž vyhodnocení dojde na základě jedné zprávy z chytrého zařízení – 3^{12} . Propustnost služby po optimalizaci tak činila 519 výsledků vyhodnocení klíčových indikátorů výkonnosti za sekundu, což představovalo 81% nárůst oproti stavu před optimalizací. I přes tuto optimalizaci zůstala, s ohledem na nezbytnost určitého počtu „pomalých“ databázových operací, propustnost služby **Backend core** nejnižší – o řád nižší než propustnost zbylých dvou služeb.

7.3 Zvyšování propustnosti systému škálováním

Podstatnou a doposud nezodpovězenou otázkou bylo, jestli a o kolik by bylo možné propustnost systému zvýšit pomocí horizontálního škálování služby **Message processing unit**. Na základě předchozích měření lze říci, že nebylo – úzké hrdlo systému představuje služba **Backend core**, resp. ukládání výsledků zpracování dat z chytrých zařízení do databáze, nikoliv samotné zpracování těchto dat. I tak je ale na místě zjistit, jestli lze škálováním služby **Message processing unit** dosáhnout vyšší propustnosti konkrétního segmentu pomyslné *pipeline* systému, tj. segmentu zpracování dat, a jestli bude nárůst propustnosti tohoto segmentu proporcionální k počtu spuštěných instancí **Message processing unit**. Pro měření jsem použil jiný scénář – na základě každého požadavku na vyhodnocení klíčových indikátorů výkonnosti dojde k vyhodnocení nikoliv 3, ale pouze jedné středně složité definice, viz výše, i jiný hardware – namísto mého notebooku disponujícího procesorem Intel® Core™ i7-12700H jsem použil propůjčený server disponující procesorem Intel® Xeon® Processor E5-2699 v3. Nešlo tak o distribuovaný hardwarový systém zahrnující zcela oddělené stroje, neboli uzly, který v kontextu horizontálního škálování

¹²Za normálních okolností se nejedná o konstantu a přínos optimalizace tak není konstantní.

zmiňuje podkapitola 3.2, ale o určitou, dostupnější náhradu výše uvedeného. Stejná zůstala metoda měření propustnosti, kterou popisuje sekce 7.1.3, tzn. měření propustnosti na bázi poklesu množství zpráv ve frontě `kpi-fulfillment-check-requests`. Důležité je také to, že měření probíhalo po provedení optimalizace popsané v podkapitole 7.2, založené na odstranění kontraproduktivního paralelismu, u kterého režie převýšila přínos paralelního vyhodnocení. Propustnost jedné instance služby během tohoto měření činila „pouze“ necelých 1200 zpráv, resp. požadavků na vyhodnocení klíčových indikátorů výkonnosti, za sekundu, což přičítám mimo jiné nižšímu výkonu jednoho jádra u procesoru serveru. Tato hodnota byla nicméně použita jako referenční hodnota pro další měření. Principem tohoto měření bylo postupné zvyšování počtu spuštěných instancí služby `Message processing unit` po mocninné řadě čísla 2, tedy 1, 2, 4, 8 atd., a měření propustnosti celého segmentu pipeline. Naměřenou propustnost segmentu jsem následně srovnal s proporcionálním nárůstem propustnosti, tedy násobkem propustnosti jedné instance a počtu spuštěných instancí služby `Message processing unit`, což ukazuje graf na obrázku.



Obrázek 7.4: Nárůst propustnosti segmentu zpracování dat s ohledem na počet instancí služby `Message processing unit`: proporcionální nárůst × realita

Z výše uvedeného grafu lze, zejména s ohledem na použití logaritmického měřítky, vyčíst, že zvyšování počtu instancí služby `Message processing unit` má z hlediska propustnosti segmentu zpracování dat smysl pouze do určitého bodu. Za tímto bodem, resp. nad tento počet instancí, se přínos škálování ztrácí – propustnost se ještě mírně zvýší, ale ne proporcionálně k počtu instancí. To je patrně způsobeno tzv. *spory o procesor*, z angl. *CPU contention*, což je situace, kdy procesor nedisponuje dostatečnými zdroji z hlediska potřeb spuštěných procesů, které se tak začnou o zdroje procesoru „přetahovat“. Z toho lze usuzovat, že optimální počet spuštěných instancí služby `Message processing unit` bude záviset na objemu zdrojů procesoru zařízení, na kterém systém, či tento konkrétní segment pipeline, poběží. Půjde-li o „obstojný“ hardware, optimální počet bude > 1 , což dokazuje smysluplnost existence služby `Message processing unit`, coby plnohodnotné komponenty architektury systému, s jasným přínosem. Opomenout nelze ani potenciální rozvoj systému zahrnující provádění výpočetně náročnějších operací nad daty z chytrých zařízení, či nasazení systému, popř. jednotlivých segmentů pipeline, na několik různých,

různě výkonných zařízení. V těchto případech bude mít škálovatelnost služby **Message processing unit** ještě vyšší význam.

7.4 Celosystémová latence

Kromě propustnosti jsem se zabýval i celosystémovou latencí, tedy tím kolik času zabere, než zpráva „projde systémem“ a promítne se do uživatelského rozhraní. Toto měření, opět založené na dvou sondách, bylo poměrně přímočaré. První sondu, jejímž účelem bylo zaznamenat časy příjetí zpráv, jsem umístil do kódu služby **MQTT preprocesor**, konkrétně pak na začátek těla funkce **addIncomingMQTTMessageToFIFO**. Druhou sondu, jejímž účelem bylo zaznamenat časy, kdy se zprávy promítnou do uživatelského rozhraní systému, pak do kódu klientské strany systému – přesněji do těla speciální funkce **useEffect**, která integruje data přicházející skrze GraphQL subscription se současným datovým modelem klientské strany. Vzhledem k omezení sondy na klientské straně, opírající se o příkaz **Date.now()**¹³, byla přesnost obou sond omezena na milisekundy. Průměrná hodnota celosystémové latence ze 100 měření činila 75 ms.

7.5 Zhodnocení

Na základě měření propustnosti jednotlivých služeb, optimalizace propustnosti služeb **Message processing unit** a **Backend core** a měření přínosu horizontálního škálování služby **Message processing unit** lze poznamenat, že systém by měl, i přes výrazně nižší propustnost služby **Backend core**, za běžných okolností obstát. Za běžné okolnosti lze, podle mých informací, označit situaci, kdy jsou k systému připojeny tisíce zařízení, ale perioda zasílání zpráv u nich činí několik minut a zátěž je tak oproti scénářům použitým při měření spíše nižší. Např. pro 6000 chytrých zařízení, která zasílají zprávy co 5 minut činí průměrná zátěž systému pouhých 20 zpráv za sekundu. To je 5krát méně, než kolik činí obezřetný odhad propustnosti služby **Backend core** – úzkého hrdla systému. Připojených chytrých zařízení by tak mohlo být až 5krát tolik – přibližně 30 000, přičemž teoretický limit zbylých služeb činí ještě 100krát tolik – tedy až 3 miliony připojených zařízení. Důležitou roli z hlediska reálných čísel pak sehraje také objem systémových zdrojů, které bude mít systém po nasazení k dispozici.

Co se týče průměrné celosystémové latence, lze říci, že z pohledu lidských uživatelů představuje naměřených 75 ms zanedbatelné, potažmo téměř nepostřehnutelné zpoždění, a je tedy možné hovořit o okamžitém zpracování dat z chytrých zařízení či zpracování dat z chytrých zařízení v reálném čase.

¹³Skupina TC39 připravuje nové oficiální rozhraní pro práci s časem v jazyce JavaScript s označením **Temporal**, které by mělo nahradit současné zastaralé rozhraní **Date**. Více viz <https://tc39.es/proposal-temporal/docs/index.html>.

Kapitola 8

Závěr

Cílem práce bylo na základě požadavků, získaných průzkumem relevantních oblastí a materiálů a rozбором současných řešení a jejich nedostatků, navrhnout nový, výkonný informační systém pro zpracování dat z chytrých zařízení, který zároveň poslouží jako referenční bod tvůrcům dalších aplikací, systémů a platforem určených pro zpracování dat z chytrých zařízení. Dalším z cílů pak bylo prozkoumat možnosti sledování chytrých zařízení a transformace dat z těchto zařízení na prakticky uplatnitelné poznatky pomocí klíčových indikátorů výkonnosti.

Výsledky práce, dle mého, s touto specifikací cílů práce do značné míry korespondují. Co se týče klíčových indikátorů výkonnosti, zjistil jsem, že je lze využít jako nástroj pro reprezentaci stavu zařízení, snadnou interpretaci hodnot obsažených v datech, identifikaci trendů a propojení dat s cíli uživatelů či strategickými cíli organizací, což blíže popisuje sekce 2.3.1. Při rozboru současného software pro zpracování dat z chytrých zařízení jsem zase narazil na určité nedostatky, kterými byly zejména omezená stabilita, omezená kapacita a škálovatelnost a absence nástrojů umožňujících proudové zpracování dat, jehož přednostmi jsou mimo jiné rychlejší propagace výsledků zpracování dat k uživatelům, či dalším připojeným systémům a aplikacím, a rozložení zátěže v čase, což může zvýšit kapacitu systému či platformy.

Výše popsané nedostatky mi spolu s rozбором potenciálních uživatelů a jejich potřeb posloužily jako předobraz požadavků na nový systém, které jsem posléze ještě upřesnil a také rozšířil o doplňující požadavky od výzkumné skupiny zabývající se problematikou chytrých měst na Ústavu informačních systémů Fakulty informačních technologií a společnosti Logimic. Popsal jsem i případy užití systému. Na tomto základě jsem poté navrhl a implementoval nový, snadno rozšiřitelný systém provádějící proudové zpracování dat z chytrých zařízení, přičemž podstatou zpracování dat je vyhodnocování uživatelských kritérií – terminologicky klíčových indikátorů výkonnosti. Výsledkem je kontinuální zpracování značného objemu dat z chytrých zařízení do zcela srozumitelné podoby – je konkrétní kritérium, např. přijatelné rozmezí teploty u teplotních čidel, splněno, či nikoliv? Systém nabízí také agregaci výsledků zpracování dat – splnění kritérií lze pozorovat nejen u konkrétních zařízení, ale také u uživatelem definovaných skupin zařízení.

Co se týče výkonnosti, kapacity a stability – systém by měl za běžných okolností obstát, ale je zde patrný prostor pro zlepšení. Důkladné měření ukázalo, že ukládání výsledků zpracování dat z chytrých zařízení do relační databáze PostgreSQL je časově náročnější, než samotné zpracování dat, na jehož optimalizaci jsem kladl od začátku důraz. Stabilitu systému lze zajistit na úrovni technologie použité pro komunikaci mezi službami systému – RabbitMQ. Nebude-li systém, potažmo konkrétní služba stíhat, technologie

RabbitMQ může začít zahazovat přebytečné zprávy, což zabrání potenciálnímu zahlcení systému. Tvůrcům dalších aplikací, systémů a platforem určených pro zpracování dat z chytrých zařízení nicméně doporučuji zaměřit se na výkonnostní charakteristiky zvolené databáze, její škálovatelnost atd. Při případné další optimalizaci tohoto systému lze zvážit použití systému CockroachDB¹, či některé z obecně lépe horizontálně škálovatelných NoSQL a NewSQL databází. Za zmínku stojí také další rozdělení zbylé funkcionality služby **Backend core** a další optimalizace z hlediska souběžnosti a paralelismu.

I přes určité potíže s omezenou rychlostí ukládání dat do databáze je podle mě systém pro zpracování dat z chytrých zařízení s označením **RIoT**², který je dostupný na adrese <https://github.com/MichalBures-OG/RIoT-IS> pod licencí MIT License³, slibným softwarovým projektem a tato práce užitečným referenčním bodem pro kohokoliv, kdo se bude zabývat zpracováním dat z chytrých zařízení, monitoringem a reprezentací stavu chytrých zařízení, informačními systémy, či výkonnostní optimalizací software – především pak informačních systémů a webových služeb.

¹Viz <https://www.cockroachlabs.com/>.

²Information system for **Real-time IoT** data processing and state monitoring through user-defined criteria – KPIs

³Viz <https://choosealicense.com/licenses/mit/>.

Literatura

- [1] ARAUJO, V., MITRA, K., SAGUNA, S. a ÅHLUND, C. Performance evaluation of FIWARE: A cloud-based IoT platform for smart cities. *Journal of Parallel and Distributed Computing*. Elsevier. 2019, sv. 132, s. 250–261.
- [2] ASHTON, K. That ‘Internet of Things’ Thing. *RFID Journal*. 2009.
- [3] BALASUBRAMANIAN, A., BARANOWSKI, M. S., BURTSEV, A., PANDA, A., RAKAMARIĆ, Z. et al. System programming in rust: Beyond safety. In: *Proceedings of the 16th workshop on hot topics in operating systems*. 2017, s. 156–161.
- [4] BAUER, M., SANCHEZ, L. a SONG, J. IoT-enabled smart cities: Evolution and outlook. *Sensors*. MDPI. 2021, sv. 21, č. 13, s. 4511.
- [5] BELLINI, P., NESI, P., PAOLUCCI, M. a ZAZA, I. Smart city architecture for data ingestion and analytics: Processes and solutions. In: IEEE. *2018 IEEE Fourth International Conference on Big Data Computing Service and Applications (BigDataService)*. 2018, s. 137–144.
- [6] BUNTZ, B. *The Top 20 Industrial IoT Applications* [online]. 2017 [cit. 2024-01-06]. Dostupné z: <https://www.iiotworldtoday.com/iiot/the-top-20-industrial-iiot-applications>.
- [7] CHANG, J. C.-J. a KING, W. R. Measuring the performance of information systems: A functional scorecard. *Journal of Management Information Systems*. Taylor & Francis. 2005, sv. 22, č. 1, s. 85–115.
- [8] CHUA, Y. W. Appreciating rust’s memory safety guarantees. *Government Digital Services, Singapore*. 2017.
- [9] COX BUDAY, K. *Concurrency in Go: Tools and Techniques for Developers*. "O’Reilly Media, Inc.", 2017.
- [10] FOWLER, M. *Patterns of Enterprise Application Architecture*. Pearson Education, 2012. Addison-Wesley Signature Series (Fowler). ISBN 9780133065213.
- [11] FOWLER, S. *Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization*. O’Reilly Media, 2016. ISBN 9781491965948.
- [12] GREENGARD, S. *The Internet of Things, revised and updated edition*. MIT Press, 2021. ISBN 9780262542623.
- [13] GREGG, B. *Systems Performance: Enterprise and the Cloud*. 2. vyd. Pearson, 2020. ISBN 9780136820154.

- [14] GRIFFITHS, F. a OOI, M. The fourth industrial revolution - Industry 4.0 and IoT [Trends in Future I&M]. *IEEE Instrumentation & Measurement Magazine*. 2018, sv. 21, č. 6, s. 29–43. DOI: 10.1109/MIM.2018.8573590.
- [15] GRIGORIK, I. *High Performance Browser Networking: What Every Web Developer Should Know about Networking and Web Performance*. O'Reilly Media, 2013. ISBN 9781449344740.
- [16] HALLER, S. The things in the internet of things. *Poster at the (IoT 2010)*. Tokyo, Japan, November. 2010, sv. 5, č. 8, s. 26–30.
- [17] HARCHOL BALTER, M. *Performance Modeling and Design of Computer Systems: Queuing Theory in Action*. Cambridge University Press, 2014. ISBN 9781139625715.
- [18] HARRISON, C., ECKMAN, B., HAMILTON, R., HARTSWICK, P., KALAGNANAM, J. et al. Foundations for Smarter Cities. *IBM Journal of Research and Development*. 2010, sv. 54, č. 4, s. 1–16. DOI: 10.1147/JRD.2010.2048257.
- [19] ISO/IEC JTC 1/SC 31. *ISO/IEC 18000-7:2014*. Standard. International Organization for Standardization, 2014. Dostupné z: <https://www.iso.org/standard/57336.html>.
- [20] ISO/IEC JTC 1/SC 31. *ISO/IEC 29167-10:2017*. Standard. International Organization for Standardization, 2017. Dostupné z: <https://www.iso.org/standard/69410.html>.
- [21] ISO/IEC JTC 1/SC 31. *ISO/IEC 18000-63:2021*. Standard. International Organization for Standardization, 2021. Dostupné z: <https://www.iso.org/standard/78309.html>.
- [22] ISO/IEC JTC 1/SC 31. *ISO/IEC 20248:2022*. Standard. International Organization for Standardization, 2022. Dostupné z: <https://www.iso.org/standard/81314.html>.
- [23] ISO/IEC JTC 1/SC 6. *ISO/IEC 18092:2023*. Standard. International Organization for Standardization, 2023. Dostupné z: <https://www.iso.org/standard/82095.html>.
- [24] KARLSSON, P. *Performance evaluation for choosing between Rust and C++*. Karlskrona, Sweden, 2023. Bakalářská práce. Blekinge Institute of Technology.
- [25] KARWATKA, P. *Monolithic architecture vs microservices* [online]. 2020 [cit. 2024-03-08]. Dostupné z: <https://www.divante.com/blog/monolithic-architecture-vs-microservices>.
- [26] KIRIMTAT, A., KREJCAR, O., KERTESZ, A. a TASGETIREN, M. F. Future Trends and Current State of Smart City Concepts: A Survey. *IEEE Access*. 2020, sv. 8, s. 86448–86467. DOI: 10.1109/ACCESS.2020.2992441.
- [27] KLEPPMANN, M. *Designing Data-Intensive Applications. The Big Ideas Behind Reliable, Scalable and Maintainable Systems*. O'Reilly, 2017. ISBN 9781449373320.
- [28] KONDEPUDI, S. N., RAMANARAYANAN, V., JAIN, A., SINGH, G. N., AGARWAL, N. N. et al. *Smart Sustainable Cities: An Analysis of Definitions*. International Telecommunication Union, říjen 2014.

- [29] LAVALLE, A., TERUEL, M. A., MATÉ, A. a TRUJILLO, J. Improving Sustainability of Smart Cities through Visualization Techniques for Big Data from IoT Devices. *Sustainability*. 2020, sv. 12, č. 14. DOI: 10.3390/su12145595. ISSN 2071-1050. Dostupné z: <https://www.mdpi.com/2071-1050/12/14/5595>.
- [30] LOGIMIC. *ACADA – rozšiřitelná platforma pro vaše produkty* [online]. 2021 [cit. 2024-01-17]. Dostupné z: <https://www.logimic.com/cs/platforma/>.
- [31] NAIK, N. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In: IEEE. *2017 IEEE international systems engineering symposium (ISSE)*. 2017, s. 1–7.
- [32] PAPPAS, L. a WHITMAN, L. Riding the technology wave: Effective dashboard data visualization. In: Springer. *Human Interface and the Management of Information. Interacting with Information: Symposium on Human Interface 2011, Held as Part of HCI International 2011, Orlando, FL, USA, July 9-14, 2011, Proceedings, Part I*. 2011, s. 249–258.
- [33] PARMENTER, D. *Key Performance Indicators: Developing, Implementing, and Using Winning KPIs*. 4th. John Wiley & Sons, 2019. ISBN 9781119620778.
- [34] RAINER, R., PRINCE, B. a CEGIELSKI, C. *Introduction to Information Systems: Supporting and Transforming Business*. Wiley, 2013. ISBN 9781118674369.
- [35] RAJAN, R. A. P. Serverless Architecture - A Revolution in Cloud Computing. In: *2018 Tenth International Conference on Advanced Computing (ICoAC)*. 2018, s. 88–93. DOI: 10.1109/ICoAC44903.2018.8939081.
- [36] SANTANA, E. F. Z., CHAVES, A. P., GEROSA, M. A., KON, F. a MILOJICIC, D. S. Software platforms for smart cities: Concepts, requirements, challenges, and a unified reference architecture. *ACM Computing Surveys*. ACM New York, NY, USA. 2017, sv. 50, č. 6, s. 1–37.
- [37] SCHILL, M., NANZ, S. a MEYER, B. Handling parallelism in a concurrency model. In: Springer. *Multicore Software Engineering, Performance, and Tools: International Conference, MUSEPAT 2013, St. Petersburg, Russia, August 19-20, 2013. Proceedings*. 2013, s. 37–48.
- [38] STILLER, B., SCHILLER, E., SCHMITT, C., ZIEGLER, S. a JAMES, M. An overview of network communication technologies for IoT. *Handbook of Internet-of-Things*. Springer Int. 2020, sv. 12.
- [39] VENUTA, M., ZAMBETTI, M., PIROLA, F., PEZZOTTA, G., GRASSEN, P. et al. Implementation of IoT Platform's Dashboards for the Visualisation of Dynamic KPIs: Insights from a Case Study. In: CAMARINHA MATOS, L. M., BOUCHER, X. a AFSARMANESH, H., ed. *Smart and Sustainable Collaborative Networks 4.0*. Cham: Springer International Publishing, 2021, s. 517–525. ISBN 978-3-030-85969-5.
- [40] ŠULC, O. *Optimalizace zpracování klíčových indikátorů výkonu*. Brno, Česká republika, 2023. Diplomová práce. Vysoké učení technické v Brně.