VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ZPRACOVÁNÍ A UKLÁDÁNÍ IOT DAT DO RELAČNÍ DATABÁZE S VYUŽITÍM CLOUDOVÝCH SLUŽEB

PROCESSING AND STORING IOT DATA IN RELATIONAL DATABASES WITH APPLICATION OF CLOUD SERVICES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JAKUB HLAVA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JIŘÍ HYNEK, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Hlava Jakub**
Program: Informační technologie
Název: **Zpracování a ukládání IoT dat do relační databáze s využitím cloudových služeb**
Processing and Storing IoT Data in Relational Databases with Application of Cloud Services
Kategorie: Databáze

Zadání:

1. Prostudujte způsoby komunikace IoT zařízení s cloudem a existujících datových modelů IoT zařízení v cloudu.
2. Prostudujte principy a technologie relačních databází a datové abstrakce (např. objektově relačního mapování). Prozkoumejte dostupné běhové prostředí pro práci s relačními databázemi (např. Node.js, Python, apod.).
3. Proveďte analýzu současného stavu dostupných softwarových řešení, které řeší datovou abstrakci v cloudu se zaměřením na relační databáze typu SQL, definujte výhody a nevýhody použití relační SQL databáze versus ne-SQL databáze.
4. Vyberte z používaných technik pro simulaci konkurenčních přístupů do SQL databáze, případně navrhněte vhodný vlastní algoritmus.
5. Dle návrhu implementujte skript (ve vybrané technologii z bodu 2) simulující souběžný přístup do relační databáze (prostřednictvím AWS lambda funkcí vyvolaných z front-end části systému, MQTT zpráv ze senzorů a periodických procedur AWS).
6. Proveďte simulaci prostřednictvím implementovaného skriptu a sledujte různé úrovně zatížení. Zajistěte detekci chyb, jejich zaznamenávání a vhodně vizualizujte zatížení. Proveďte výsledné zhodnocení.

Literatura:

- Greengard, S.: *The Internet of Things*. MIT Press, 2015, ISBN 978-026-2527-736.
- The PostgreSQL Global Development Group: *Documentation* [online]. 2021 [cit. 2021-10-09]. Dostupné z: https://www.postgresql.org/docs/
- TypeORM: *TypeORM* [online]. 2021 [cit. 2021-10-09]. Dostupné z: https://typeorm.io/
- Amazon Web Services, Inc.: *AWS Documentation* [online]. 2021 [cit. 2021-10-09]. Dostupné z: https://docs.aws.amazon.com/index.html
- Interní dokumentace firmy Logimic.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 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. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 25. října 2021

Abstrakt

Chytrá města a systémy pro jejich správu neustále generují velké množství dat, která je nutné zpracovávat a ukládat. Cílem této bakalářské práce je vytvořit nástroj, který vývojářům systému pro správu chytrých měst umožní simulovat vliv tohoto množství dat na jejich systém a především na databázový server. Díky tomu budou moci přizpůsobit používanou cloudovou infrastrukturu předpokládané zátěži ještě před ostrým nasazením. Nástroj je vyvinut v jazyce Python jako soubor komponent, které umožňují škálovatelné zátěžové testování z mnoha míst současně. Během testování nástroj sbírá data o jeho průběhu a o provozu databáze PostgreSQL, díky kterým lze pozorovat vliv zatížení na testovaný systém. Průběh testování uživatel řídí a monitoruje pomocí integrovaného webového rozhraní, které navíc vizualizuje průběžně nasbíraná data v podobě grafů. Nástroj vznikl ve spolupráci s firmou Logimic, s.r.o., na míru jejich systému pro správu chytrých měst.

Abstract

Smart cities and systems for their management are constantly producing great amount of data, which needs to be processed and stored. The goal of this Bachelor's thesis is to create a tool, which will make developers able to test influence of this amount of data on their system, especially on database server. Then they will be able to adjust their cloud infrastructure for assumed level of load before it's real-world deployment. The tool is developed using Python programming language as collection of components which makes scalable load testing possible from many places at the same time. Data about the course of test and about service of PostgreSQL database server is collected by the tool during testing. Course of the test can be controlled and monitored by user using integrated web interface, which also visualizes collected data in form of charts. The tool was created in cooperation with Logimic, s.r.o., tailored to their system for smart cities management.

Klíčová slova

databáze, chytrá města, zátěžový test, internet věcí, IoT, AWS, cloud, PostgreSQL, Python

Keywords

database, smart cities, load test, internet of things, IoT, AWS, cloud, PostgreSQL, Python

Citace

HLAVA, Jakub. *Zpracování a ukládání IoT dat do relační databáze s využitím cloudových služeb*. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Hynek, Ph.D.

Zpracování a ukládání IoT dat do relační databáze s využitím cloudových služeb

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Hynka, Ph.D. Další informace mi poskytla firma Logimic, s.r.o. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jakub Hlava
9. května 2022

Poděkování

Chtěl bych v prvé řadě poděkovat vedoucímu práce, Ing. Jiřímu Hynkovi, Ph.D. za pomoc a podporu, četné rady a časté konzultace, které mi v průběhu vypracování práce poskytoval. Dále bych chtěl poděkovat Ing. Františkovi Mikulů, CTO firmy Logimic, s.r.o., za odborné konzultace, technickou podporu a poskytnuté materiály. V závěru bych chtěl poděkovat své rodině a přátelům za podporu a oporu v průběhu celého studia a především během psaní této práce.

Obsah

Kapitola 1

Úvod

Chytrá města množstvím připojených zařízení a senzorů do internetu věcí generují obrovské množství dat. Řádově se v každém městě jedná o miliony senzorů a zařízení generující data 24 hodin denně, 7 dní v týdnu. Tato data je nutné spolehlivě sbírat, zpracovávat a analyzovat, a to prakticky v reálném čase, aby další zařízení v síti chytrého města mohla na tato data adekvátně reagovat a poskytovat lidem ve městě požadované služby, stejně jako včas informovat techniky, údržbu, záchranné složky nebo vedení města o detekovaných problémech a ochránit včasnou reakcí zdraví a majetek. Takové množství uživatelů a zařízení, které může u měst dosahovat řádově i milionů a dat, která jsou zařízeními a uživateli generována a je nutné je zpracovávat, vytváří vysoké nároky na systém, který sběr, zpracování a prezentaci dat zajišťuje. Pokud tento systém trpí slabými místy, může při sběru a zpracování docházet ke zpomalení činnosti nebo dokonce k odmítnutí a zahození požadavků a s tím související ztrátu dat a z toho dále vyplývající potíže.

Cílem této práce je vytvořit software, který bude simulovat množství zařízení, senzorů a uživatelů přistupující k systému pro sběr, zpracování a analýzu dat z IoT zařízení v jeden čas a souběžně bude monitorovat a zaznamenávat chování systému, časy přístupů a konzistenci uložených dat. Tato data poté bude vizualizovat pro usnadnění identifikace slabých míst v systému.

Teoretická část této práce se skládá ze 2 kapitol. Kapitola 2 je úvodem do internetu věcí a obsahuje základní rozbor technologií IoT zařízení, principů jejich komunikace s cloudem a cloudových řešení využitých v praktické části. V kapitole 3 jsou rozebrány technologie a principy relačních i nerelačních databází, použití objektově relačního mapování (ORM) pro práci s nimi při implementaci softwaru a rozebrány možnosti nasazení databázových systémů v cloudovém prostředí.

V praktické části je v kapitole 4 provedena analýza aktuálního stavu, požadavků a průzkum trhu. Kapitola 5 popisuje návrh způsobu testování systému pro správu chytrých měst a samotné aplikace pro testování. V kapitole 6 jsou popsány detaily implementace testovací aplikace. Výstupy z používání aplikace a další detaily testování popisuje kapitola 7. V závěrečné kapitole 8 je provedeno zhodnocení výsledků provedených simulací.

Kapitola 2

Úvod do internetu věcí, komunikace a cloudu

Tato kapitola rozebírá význam a problematiku internetu věcí (v části 2.1), cloudu a cloudových služeb, především v kontextu IoT (sekce 2.2), dále zmiňuje technologické principy IoT zařízení (sekce 2.3) a hlouběji se věnuje protokolu MQTT (část 2.3.1).

2.1 Internet věcí

Pojem *internet věcí* (nebo také angl. *Internet of Things*, zkráceně IoT) definuje Oxfordský slovník jako propojení zařízení uvnitř běžných předmětů skrze internet, které jim umožňuje sdílet data [22]. Alternativní, byť podobnou definici nabízí slovník Merriam-Webster, ten internet věcí definuje jako síťovou schopnost, umožňující informacím, aby byly odesílány a přijímány napříč objekty a zařízeními (např. příslušenstvím v domě nebo kuchyňskými spotřebiči) za použití Internetu [33]. Na základě těchto definic můžeme zjednodušeně říci, že za zařízení internetu věcí můžeme považovat prakticky jakékoliv běžné zařízení (např. žárovku, vypínač, lednici, termostat, teploměr, klimatizaci, ...), kterému byla za pomoci dalších součástí (viz obrázek 2.1) přidána schopnost připojit se k Internetu a skrze Internet přijímat a odesílat informace (teplotu, stav – svítí/nesvítí, příkazy – rozsviť/zhasni) a internetem věcí rozumíme síť, která takto rozšířená zařízení spojuje.

2.1.1 Kategorizace IoT zařízení dle schopností

Podle článku [38] lze zařízení internetu věcí, resp. „chytrá, propojená zařízení“ rozdělit do čtyř skupin podle jejich schopností. Funkcionalita každé ze skupin zahrnuje i funkcionalitu všech předcházejících. Jde postupně o schopnosti monitorování, ovládání, optimalizace a autonomie.

Schopnost *monitorování* znamená, že je zařízení vybaveno senzory a součástmi, které jsou schopny sledovat stav tohoto zařízení, stav jeho okolí a dokáže za základě těchto stavů zasílat upozornění ostatním zařízením nebo uživatelům, tato data lze též ukládat a vyhodnocovat na jejich základě návrh, činnost a průběh života tohoto zařízení.

Zařízení se schopností *ovládání* je možné řídit. Řízení probíhá buď dálkově pomocí zasílání příkazů nebo pomocí algoritmů, které řídí chování automaticky podle sady pravidel a aktuálního stavu, který je získán zmíněným monitorováním. Pod zasláním příkazu si lze představit např. zmáčknutí tlačítka v mobilní aplikaci, která následně zašle zprávu zařízení,

pod řídícím algoritmem např. postup a pravidla pro automatické otevření kohoutu, pokud bude tlak v monitorované nádrži příliš vysoký.

Optimalizace je schopnost, při které se využívá kombinace proudu monitorovacích dat ze zařízení společně se schopností tato zařízení ovládat a řídit. Toto spojení umožňuje zařízením využít algoritmy, aktuální a historická data k tomu, aby automaticky upravily svá nastavení k dosažení dramatických nárůstů výstupů, účinnosti a využití. Dále lze za pomoci této kombinace schopností zoptimalizovat údržbu a opravy zařízení, protože o jejich funkci, výkonu a opotřebení mají jejich majitelé díky proudu dat přesné informace a mohou určité problémy vyřešit dálkově, předejít jim úpravou nastavení nebo včas vyslat k zařízení techniky.

Autonomie vzniká kombinací všech tří předcházejících schopností. Zařízení jsou na základě znalosti svého stavu, možnosti sama sebe řídit a schopnosti optimalizovat svoji činnost, schopna jednat na určité úrovni samostatnosti a komunikovat a koordinovat se s dalšími zařízeními a systémy. Uživatel poté zařízením nebo jejich skupinám pouze zadává práci a kontroluje výsledky.

2.1.2 IoT v každodenním životě

Internet věcí zasahuje do našeho každodenního života již nyní, jakkoliv se jedná o poměrně nový princip, rozsah, jak nás IoT ovlivňuje, ilustruje úvod první kapitoly knihy [19]. Autor v této kapitole popisuje svůj běžný den (již v roce 2015), při čemž klade důraz především na elektroniku a prvky internetu věcí, se kterými denně operuje.

Autor se například každé ráno váží, ale namísto vedení si záznamů o tělesné váze na papíře se o zaznamenání jeho hmotnosti stará jeho osobní váha vybavená procesorem a připojením k Internetu. Svoji aktuální hmotnost včetně jejího vývoje v minulosti potom může vidět v mobilní aplikaci. Podobně autor namísto ručního vedení knihy jízd a spotřeby paliva využívá doplněk do auta, který tato data sbírá automaticky, ukládá je a přes Internet odesílá k uložení a automatickému zpracování. Společným jmenovatelem těchto příkladů je pak zbavení uživatele časově náročné a repetitivní činnosti sepisování dat a přepočítávání těchto dat na relevantní výsledky. Příslušné aplikace k těmto zařízením totiž uživateli prezentují již jasný a přehledný výsledek. Tento přehledný výsledek může být i výsledkem agregace dat z různých zdrojů, a zařízení a to obzvlášť u monitorování fitness aktivity, příkladem může být systém Google Fit [16], který sbírá data s propojených zařízení a aplikací (např. ze zmíněné chytré váhy nebo náramku měřícím tep, běžícím páse, měřícím výkony uživatele apod.) a podává uživateli ucelený obraz o jeho stavu.

Ještě lepším příkladem pro dotvoření představy o internetu věcí je automatizace a monitoring domu. Autor ve svém popisu běžného dne zmiňuje výbavu domu zámky, termostaty a světly, která jsou taktéž rozšířena procesorem a připojením k internetu. Namísto obcházení domu, aby před odchodem přenastavil termostaty mu stačí pouze přepnout systém do jiného, uloženého režimu, což nastaví správné teploty u všech termostatů v domě. Namísto hledání náhradního klíče a osobního dodání klíče sousedovi stačí pouze přidat dočasný kód od zámku dveří, aby se soused mohl pomoci postarat o rostliny nebo zvířata během dovolené. Pro ovládání světel, garážových vrat, rolet a podobných domácích zařízení, která vybavíme „chytrou“ IoT částí – procesorem a komunikačním modulem, také nemusí uživatel nikam chodit a vše může řídit z obslužné aplikace, obvykle je procesor schopný i zjistit, v jakém stavu se fyzická část zařízení nachází, takže uživatel vidí, kde se svítí nebo zda má například zavřenou garáž. Rozsvěcení světel, ovládání topení a další procesy je možné spojit s informacemi, které poskytuje internet, konkrétně například předpovědí počasí, časy

východů a západů slunce, které umožní ještě o něco zvýšit pohodlí uživatele a osvětlení a topení si plně zautomatizovat.

Na těchto příkladech lze pozorovat, že kýženým důsledkem je větší pohodlí, větší přehled, méně práce a méně starostí pro obyvatele domů se zařízeními, které lze připojit do internetu věcí. Z výčtu těchto příkladu pak logicky vyplývá, že obyvatelé domů, stejně jako uživatelé IoT doplňků např. pro fitness jsou schopni z těchto zařízení získat velké množství konkrétních dat a na jejich základě mít velmi detailní přehled o svém domě, resp. svém zdraví a výkonech. Na základě těchto dat a předmětů pak obyvatelé, mimo již zmíněné zvýšení pohodlí a ušetřený čas, najít i finanční úspory, např. za svícení a topení a obecně zefektivnit fungování domácnosti.

Analogicky k řízení domu za pomoci internetu věcí lze přistupovat i k větším objektům a celkům a obdobným způsobem monitorovat, řídit a případně zefektivňovat továrny, firmy, města, apod. Internet věcí je proto také často spojován s termínem „čtvrtá průmyslová revoluce“. Podle knihy [40] lze současnou dobu se svým rapidním rozvojem technologií v širokém spektru oborů označit za další průmyslovou revoluci, kdy po automatizaci výroby a masivním rozšíření nejprve počítačů a poté internetu, přichází rozvoj umělé inteligence, miniaturních a levných senzorů a mobilního internetu a rozvoj těchto technických prostředků umožňuje rozvoj dalších oblastí vědy a internet věcí v tomto pokroku hraje roli jednoho z hlavních mostů mezi fyzickými a digitálními aplikacemi a spojuje tyto propojené „věci“ s lidmi.

2.2 Cloud computing

Cloud computing je termín, který úzce souvisí s internetem věcí, této souvislosti se blíže věnuje kapitola 2.3. Obecně se ale podle slovníku Merriam-Webster jedná o praxi ukládání běžně používaných dat na vícero serverech, dostupných napříč internetem [32], Oxfordský slovník jej definuje jako způsob využití počítačů, ve kterém jsou data a software uloženy nebo spravovány na síti počítačů, ke kterým mají uživatelé přístup přes Internet [21].

2.2.1 Členění cloudových služeb

Aplikace a uživatelé ke cloudu přistupují pomocí řady různých služeb. Tyto služby se obecně člení do 3 hlavních modelů: SaaS (*Software-as-a-Service*), PaaS (*Platform-as-a-Service*) a IaaS (*Infrastucture-as-a-Service*).

Model *SaaS* poskytuje cílovému klientu software běžící v cloudu za pomoci uživatelského rozhraní ve webovém prohlížeči, uživatel při práci se softwarem tedy využívá prostředky (procesorový čas, paměť) poskytované cloudem a stejně tak jsou na datovém úložišti v rámci cloudu uložena data, která uživatel využívá k práci. Příkladem SaaS řešení může být webový kancelářský balík Google Docs nebo Microsoft 365.

Naproti tomu model *PaaS* neposkytuje uživateli ucelenou aplikaci, ale prostředky a technologie pro vývojáře pro běh jejich aplikací – servery, virtuální servery, operační systémy, databázová řešení, síťová řešení a umožňují vývojářům, aby se soustředili pouze na jejich vlastní aplikaci a nemuseli se starat o provoz serverového řešení. Řešení s modelem PaaS jsou například služby Microsoft Azure nebo Google App Engine. [8]

K modelu PaaS se částečně pojí pojem *serverless computing* a model FaaS (*Function-as-a-Service*). *Serverless computing* se jako pojem svým významem částečně překrývá s modelem PaaS, jde totiž o formu cloudu, který umožňuje uživatelům spouštět podrobně účtované, událostmi řízené aplikace bez nutnosti se zabývat provozní logikou. Vývojáři se soustředí

pouze na úroveň funkcí, událostí a dotazů v rámci jejich aplikace a provozovatel serverless cloudu se stará o to, aby tyto součásti byly dostupné, přičemž klient je účtován pouze za konkrétní využité systémové prostředky. Model *FaaS* je forma serverless cloudu, ve které klient dodává malé, samostatné, bezstavové funkce, spouštěné na základě událostí v systému. Provozovatel cloudu je zodpovědný za dodání prostředků a zajištění spouštění této funkce podle požadavků klienta. Jednou z platforem, poskytujících model FaaS je *AWS Lambda*, kterou blíže popisuje podkapitola 2.2.3. [44]

Nejmenší abstrakci poskytuje model *IaaS*, který klientovi poskytuje virtuální datové centrum v rámci cloudu, jsou tedy poskytovány celé servery a datová úložiště, ale oproti PaaS si musí klienti servery sami nainstalovat včetně databázového řešení, aplikačních serverů a veškerého dalšího nutného softwaru. Po instalaci zůstává celé zodpovědnost za údržbu a provoz serverů a nasazení aplikací do provozu na klientech. Příkladem IaaS architektury je služba Amazon Elastic Compute Cloud (EC2). [27]

2.2.2 Amazon Web Services

Amazon Web Services, zkráceně AWS je platforma poskytující stovky typů služeb pro provoz různých typů aplikací a jejich infrastruktur ve škálovatelném cloudu. Mezi základní služby, které mohou být relevantní i z pohledu zajištění backendu pro IoT patří například *Amazon EC2* poskytující škálovatelné virtuální servery, *Amazon S3* poskytující objektové úložiště pro příjem a ukládání jakéhokoliv množství dat, nerelační databáze *Amazon DynamoDB*, relační databáze *Amazon RDS*, již zmíněná FaaS platforma *AWS Lambda* dále rozvedená v podkapitole 2.2.3 nebo *Amazon Kinesis* pro zpracovávání proudů dat v reálném čase. [3]

2.2.3 AWS Lambda

Jedná se o službu, která umožňuje spouštět kód (funkce) pro aplikace nebo backendové služby na základě událostí bez nutnosti, aby musel uživatel pro spouštění tohoto kódu nebo aplikace vytvořit a spravovat server. Události mohou pocházet z činnosti různých dalších služeb AWS (nahrání souboru do S3, příjem požadavku na API Gateway, apod). V kontextu IoT lze tyto vlastnosti využít tak, že funkce v AWS Lambda slouží jako vstupní a výstupní rozhraní pro serverové řešení IoT sítě, kde za tímto rozhraním budou připojeny další služby jako např. databáze nebo webové rozhraní pro monitoring a správu sítě. Pro každou událost (plánovanou nebo příchozí požadavek) se funkce spouští zvlášť a s využitím cloudu se toto řešení automaticky škáluje a dokáže obsloužit jak malou IoT síť s několika zařízeními generující několik málo požadavků denně, tak i hypotetické chytré město, kde jsou zařízení miliony a požadavků na cloud i statisíce za vteřinu. [6]

2.3 Technologie IoT

Technologie internetu věcí se skládá ze tří hlavních vrstev, nejnižší vrstvou je samotné IoT zařízení, nejvyšší vrstvou jsou cloudové služby a tyto dva technologické celky jsou propojeny vrstvou konektivity.

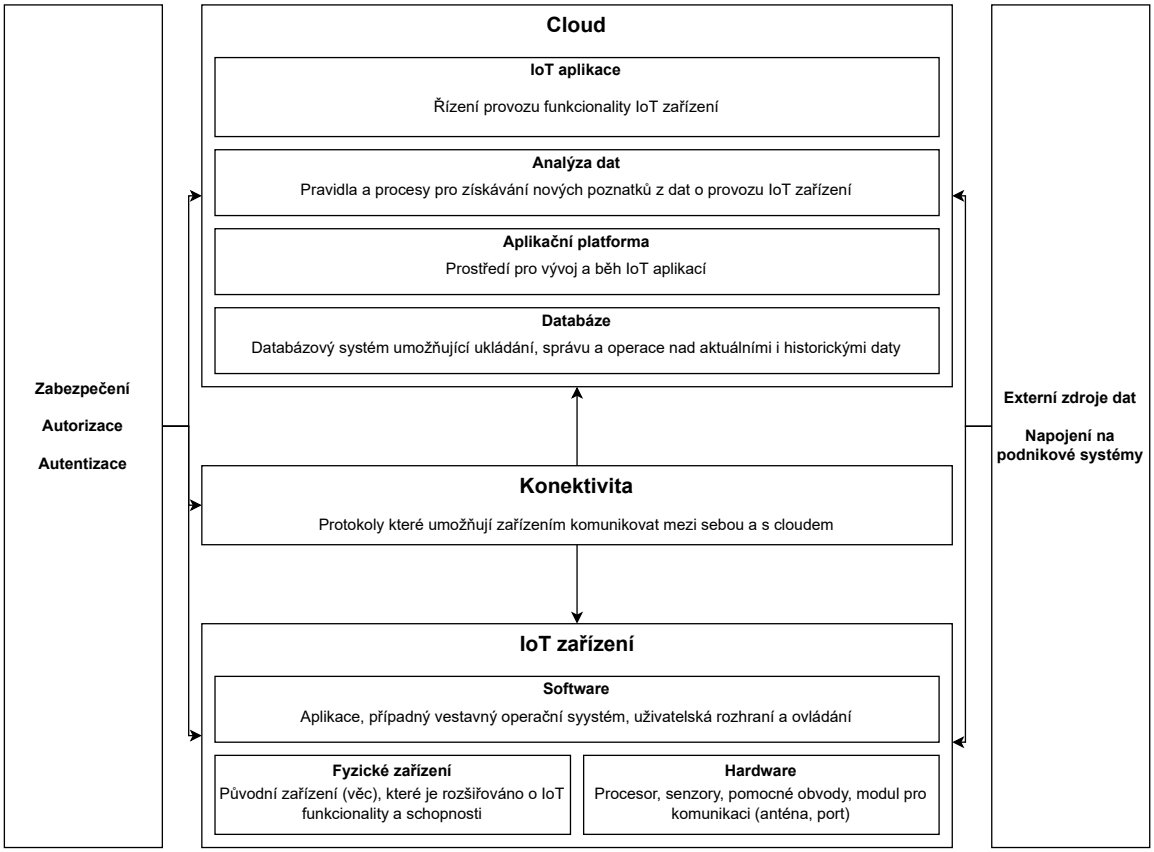
IoT zařízení se skládá ze tří primárních součástí, nejnižší z nich je fyzická část zařízení, samotná žárovka, topení, zámek nebo například motor garážových vrat. Fyzickou součást rozšiřuje chytrá část zařízení – senzory, ovládací prvky, procesory a software. Poslední důležitou součástí jsou komponenty zajišťující propojení – porty (např. Ethernet), antény (pro

Wi-Fi nebo GSM komunikaci) a taktéž musí zařízení podporovat komunikační protokoly, které specifikuje vrstva konektivity.

Vrstvu konektivity tvoří primárně protokoly, které zajišťují komunikaci zařízení s cloudem i mezi sebou, standardně například aplikační protokol MQTT blíže vysvětlený v podsekci 2.3.1.

Nejvyšší vrstva, cloudová služba má mnoho úkolů. Hlavním je příjem dat z IoT zařízení, jejich zpracování a uložení do určité formy databáze. Dále se pak cloud stará o provoz aplikací, které spravují schopnosti IoT zařízení (viz podsekci 2.1.1) a v neposlední řadě o provoz analytických a vizualizačních operací nad získanými daty, provádění operací nad daty a získávání relevantních informací z množství sesbíraných dat.

Mimo tři hlavní vrstvy se v této technologické struktuře nachází softwarové komponenty, které umožňují propojení IoT systému s jinými podnikovými systémy, případně umožňují přistupovat k externím datům – informacím o počasí, kurzech měn nebo například cenám komodit. Poslední, ale neméně důležitou součástí jsou softwarové komponenty zajišťující autentizaci, autorizaci a zabezpečení komunikace obecně. [38]



Obrázek 2.1: Struktura IoT technologie [38] [45]

2.3.1 Protokol MQTT

Standardem mezi protokoly pro IoT komunikaci je protokol MQTT[1]. Jedná se o jednoduchý protokol, navržený pro zařízení, u kterých se počítá s nekvalitním připojením, omezeními přenosu dat a vysokou odezvou, určený pro komunikaci přes centrální server, tzv. *MQTT broker*. Komunikace probíhá principem publikace zpráv k tématům (*topics*), každá zpráva zaslaná brokeru musí mít klientem přiřazeno právě jedno téma, podle kterého ji broker doručí dalším klientům.

Témata jsou organizována ve stromové struktuře, kde každé téma může obsahovat neomezený počet podtémat, jména témat proto obsahují celou cestu k tématu přes všechny úrovně stromové struktury. Jednotlivé úrovně se oddělují pomocí znaku `/`, kořenem struktury je tedy téma `/`, téma na nejvyšší úrovni pod kořenovým bude mít jméno ve formátu `/tema`, téma zanořené hlouběji ve struktuře může mít například jméno `/tema/podtema/hluboko_zanorene_tema`. Pokud má klient zájem přijímat zprávy pro dané téma nebo skupinu témat, informuje o tom broker a sdělí při tom brokeru *filtr témat*.

Filtr témat je buď konkrétní jméno konkrétního tématu nebo může mezi oddělovači `/` obsahovat tzv. *wildcard znaky* `#` a `+`. Znak `#` ve struktuře značí všechna podtémata na všech nižších úrovních, speciálně jej lze použít i samostatně (při odebírání tématu použít jako filtr `#`), což značí odběr všech témat a podtémat v celém brokeru. Znak `+` ve struktuře značí všechna témata na aktuální úrovni. [1] Při struktuře témat z obrázku 2.2 by tedy filtr `/domov/loznice/#` zajistil odběr zpráv z témat `/domov/loznice/teplota` a `/domov/loznice/topeni`, tedy všech zařízení v ložnici. Filtr `/domov/+/teplota` by zajistil odběr témat `/domov/kuchyne/teplota`, `/domov/loznice/teplota` a `/domov/obyvak/teplota`, tedy zpráv ze všech teploměrů v domě.

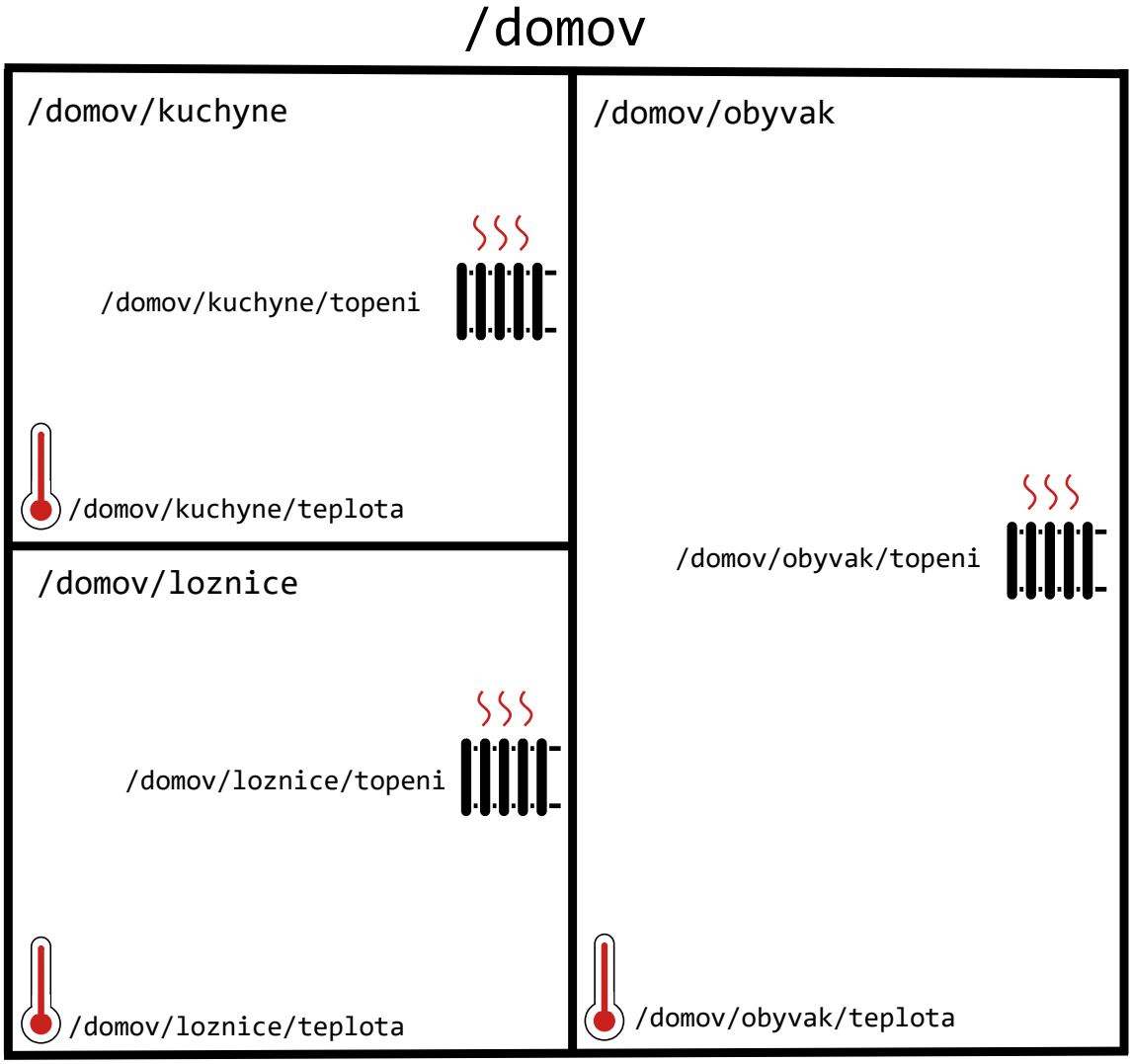
Za zabezpečení komunikace při použití protokolu MQTT je zodpovědný vývojář konkrétní aplikace při implementaci komunikace skrze MQTT. Samotný protokol umožňuje při připojování k serveru autentizovat klienta pomocí uživatelského jména a hesla (příp. tokenu nebo jiného klíče podle konkrétní implementace autentizace) a dále umožňuje MQTT komunikaci přenášet šifrovaně pomocí TLS a zajistit tak integritu a soukromí zasílaných zpráv.

Základní principy MQTT na příkladu

Na obrázku 2.2 je znázorněna část IoT systému v domě komunikující skrze MQTT, jejíž součástí jsou pouze topení a teploměry. Mimo nákres je součástí systému i cloud, který sbírá data z teploměrů a řídí topení. Stromová struktura témat je využita následovně: budovu domu reprezentuje téma `/domov` a každá místnost domu má v rámci tématu `/domov` vlastní podtéma, tedy například k ložnici patří téma `/domov/loznice`. V rámci jednotlivých místností mají IoT zařízení vlastní témata (např. `/domov/loznice/topeni` pro ovládání topení a `/domov/loznice/teplota` pro získávání teploty z teploměru v místnosti). Nastavení komunikace a samotná komunikace může v tomto případě vypadat například následovně: ovládání topení se přihlásí k odběru podtématu `topeni` v odpovídající místnosti `/domov/mistnost`, topení v ložnici tedy k tématu `/domov/loznice/topeni` a na základě příchozích zpráv upravuje intenzitu, s jakou jím řízená jednotka topí. Řídící aplikace v cloudu je přihlášena k odběru zpráv o teplotě ve všech místnostech `/domov/+/teplota`, disponuje požadovanými teplotami, na které se mají vytopit jednotlivé místnosti, ty předem pomocí rozhraní (např. webová, mobilní aplikace) zadal uživatel a může disponovat i informací o počasí v dané

[1] MQTT je standardizován neziskovou organizací OASIS, která se zabývá otevřenými standardy mj. pro kyberbezpečnost, blockchain a IoT [35]

lokalitě z internetu. Na základě těchto všech informací cloudová aplikace vyhodnotí vhodnou strategii pro dosažení požadované teploty a publikuje k tématům pro jednotlivá topení v místnostech konkrétní příkazy pro ovladače topení.



Obrázek 2.2: Nákres části IoT systému s reprezentací v MQTT brokeru

Kapitola 3

Databáze

Tato kapitola se věnuje základním pojmům a principům z oblasti relačních databází (sekce 3.1), zabývá se problematikou propojení relačně a objektově orientovaných aplikací včetně stručného popisu vybraných nástrojů (sekce 3.2), okrajově zmiňuje konkrétní software, použitý v praktické části práce (sekce 3.1.5). Dále se kapitola v sekci 3.3 věnuje popisu a výčtu základních typů nerelačních databází, jejich srovnání s relačními a popisu několika konkrétních systémů. Na závěr, v sekci 3.4 jsou popsány možnosti nasazení databázového systému na cloudové platformy i vlastní hardware.

3.1 Relační databáze

Systém řízení báze dat (zkr. SŘBD), anglicky Database Management System (zkr. DBMS) označuje podle knihy [31] software pro popis a ukládání dat a dotazování se nad nimi a to nezávisle na aplikaci. Každý SŘBD má skladovací a řídící komponentu. Skladovací komponenta obsahuje data, popisy dat a udržuje je v organizované formě, zatímco řídící komponenta obsahuje jazyk pro dotazování a manipulaci s daty a informacemi, čímž zajišťuje uživatelské rozhraní s databází, vč. autorizace.

Aby bylo možné nějakou část reálného světa reprezentovat ve formě databáze, je nutné nejprve identifikovat všechny kategorie dat, tzv. entitní množiny (např. osoby, objednávky, zboží, apod.) a jejich vzájemné vztahy (osoba má objednávku, objednávka obsahuje zboží, atd.), které budou v databázi zahrnuty, následně z nich vytvořit konceptuální model (například entitně-vztahový model) a ten transformovat do podoby databázového schématu.

3.1.1 Entitně-vztahový konceptuální model

Entitně-vztahový (zkr. ER z angl. Entity-Relationship) model slouží pro strukturování a grafickou reprezentací identifikovaných faktů o systému. Skládá se z entitních a vztahových množin.

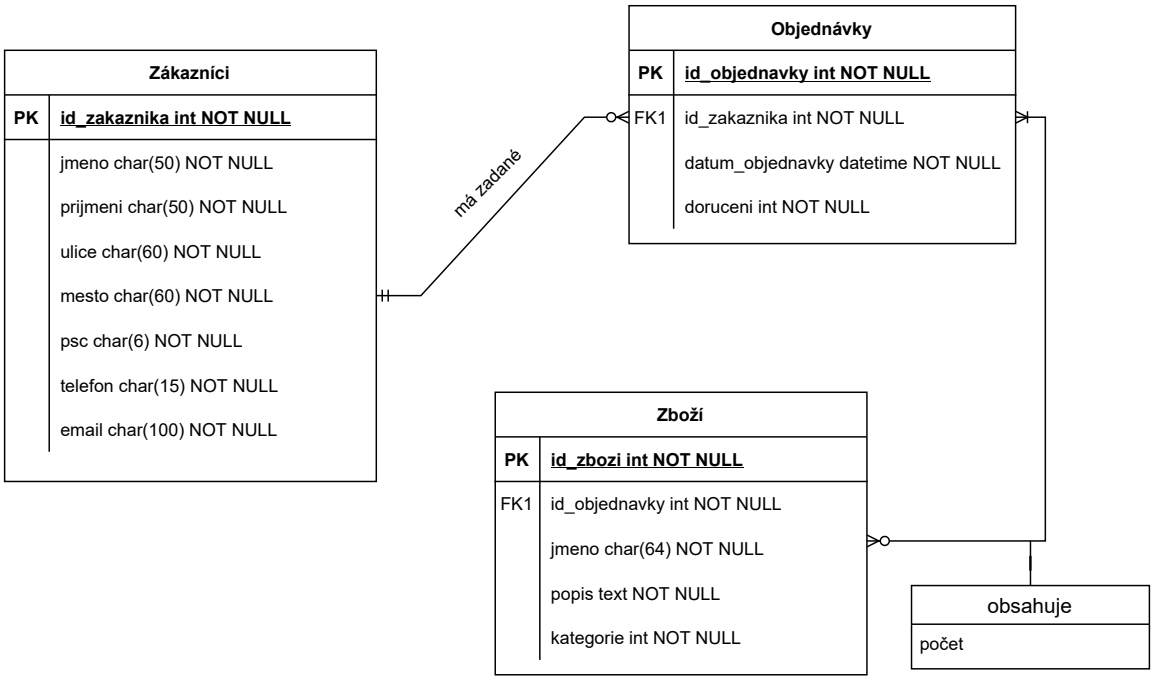
Entita je konkrétní objekt reálného světa, odlišitelný od ostatních, např. kus zboží, konkrétní osoba nebo nějaká objednávka. Entity stejných typů se sdružují do *entitních množin* a jsou charakterizovány pomocí atributů – položek v rámci entity, které specifikují např. cenu, jméno, apod. Každá entita navíc musí mít mezi atributy tzv. *identifikační klíč*, který se skládá z jednoho nebo více atributů, musí být unikátní a minimální (nesmí být možné odebrat atribut z klíče bez porušení pravidla unikátnosti).

Vztahy vznikají mezi jednotlivými entitními množinami, jsou směrové (od jedné množiny k jiné, nikoliv obecně mezi množinami) a mohou, ale nemusí obsahovat atributy, které vztah

dále specifikují. *Vztahové množiny* poté sdružují vztahy jedné entity k ostatním (např. všechny objednávky ve kterých figuruje určitý typ zboží).

Na každé straně vztahu má asociace mezi tabulkami jeden ze 4 typů – *unikátní*, kde má každá entita z jedné strany přiřazenu právě jednu entitu z druhé, *podmíněný*, kde má každá entita z jedné strany přiřazenu žádnou nebo jednu entitu z druhé, takový vztah je tedy volitelný, *mnohonásobný*, kde má každá entity z jedné strany přiřazenu jednu nebo více entit z druhé strany vztahu a *mnohonásobný podmíněný*, kde každá entita z jedné strany má přiřazenu žádnou nebo více entit z druhé strany, jedné se tedy také o volitelný vztah. Na základě těchto typů lze určit *kardinalitu* vztahu, která je párem (typ asociace první množiny k druhé, typ asociace druhé množiny k první). [31]

Na obrázku 3.1 je možné vidět jednoduchý příklad vizualizace entitně-vztahového modelu pomocí ER diagramu. V rámci diagramu jsou vizualizovány entitní množiny Zákazníci, Objednávky a Zboží. Tyto jsou vázány vztahovými množinami „má zadané“ mezi zákazníkem a objednávkami a „obsahuje“ mezi objednávkami a zbožím. Množina „obsahuje“ je navíc rozšířena atributem „počet“, který pro každý vztah z této množiny definuje množství příslušného zboží v objednávce.



Obrázek 3.1: Příklad ER diagramu

3.1.2 Schéma relační databáze

Relační databázi tvoří množina relací. *Relace* je množinou n-tic a představuje entitní množinu – abstrakci prvků reálného světa. Jednotlivé vlastnosti každé entitní množiny v podobě relace jsou popsány pomocí relačního schéma. *Relační schéma* pojmenovává relaci (např. Zákazník) a popisuje všechny součásti n-tice, která tvoří relaci (např. specifikuje, že první položkou n-tice je číslo ID, druhou řetězec Jméno, atp.). [28] Ačkoliv kniha [31] uvádí termíny relace a tabulka jako zaměnitelné, definice v knize [28] nasvědčuje tomu, že relací rozumíme matematickou podobu, zatímco tabulkou její databázovou podobu, která se skládá ze spojení relace a relačního schématu. V kontextu tabulky lze atributy a n-tice

s daty chápat jako její sloupce a řádky. Každý sloupec reprezentuje atribut tabulky (entitní množiny), ten je pojmenovaný a hodnoty v něm jsou omezeny datovým typem sloupce (např. osoba má jméno, příjmení, zboží má katalogové číslo, název, popis, počet kusů na skladě). Řádky reprezentují samotné záznamy v tabulce, konkrétní entity. Každá tabulka navíc musí obsahovat sloupec (nebo sloupce) s identifikačním (tzv. primárním) klíčem. Klíč je složen z jednoho nebo více atributů tabulky, které jednoznačně identifikují záznam na řádku, žádný řádek tedy nesmí mít stejný klíč jako jiný řádek.

Vztahové množiny mají vlastnost, *kardinalitu*, ta udává počty entit jednotlivých entitních množin, které se mohou na každé straně vztahu vyskytovat. Pro vztahovou množinu, která spojuje entitní množiny A a B můžeme rozlišit 3 kardinality. Kardinalita vztahu *1:N* (*one-to-many*) značí, že jedna entita množiny A je ve vztahu s více entitami množiny B, z pohledu množiny B bychom takový vztah označili jako *N:1* (*many-to-one*). Pokud by více entit množiny A bylo ve vztahu zároveň s více entitami množiny B, pak se tato kardinalita nazývá *M:N* (*many-to-many*). Vztah jedné entity množiny A s jednou entitou množiny B se označuje *1:1* (*one-to-one*).

V souvislosti s kardinalitami je možné pro vztahy definovat povinnost účasti entit v tomto vztahu, touto *povinností* rozumíme situaci, kdy se každá entita jedné entitní množiny musí účastnit vztahu s entitami druhé množiny. Ve výchozím stavu považujeme vztahy za nepovinné, tedy za vztahy, kterých se nemusí za každých okolností účastnit obě strany (např. při vztahu s kardinalitou 1:N mezi entitními množinami A a B může nastat situace, kdy entita množiny A nebude ve vztahu s žádnou entitou množiny B). [28]

Každá vztahová množina může být teoreticky reprezentována formou samostatné tabulky, aby ale byl výsledný počet tabulek v databázi co nejnižší, řídí se převod vztahů jejich kardinalitou. Pouze vztahy typu M:N jsou převedeny pomocí tabulky, tato tabulka obsahuje alespoň identifikační klíče obou členů vztahu, tzv. *cizí klíče*, navíc může obsahovat další atributy, které patří ke vztahové množině. Vztahy 1:N nepotřebují zvlášť tabulku, namísto toho je v tabulce s unikátním typem asociace připojen sloupec s identifikačním (*cizím*) klíčem entity na druhé straně vztahu. Shodně, přidáním sloupce s cizím klíčem, se převádí vztahy 1:1, pokud nejsou spojeny do jedné z tabulek. [31]

3.1.3 Operace nad databází

Nad takto organizovanými daty je možné provádět dotazovací a manipulační operace. Tyto operace se dělí do 4 kategorií, souhrnně označovaných zkratkou CRUD – **C**reate (vytváření), **R**ead (čtení), **U**pdate (změna), **D**elete (mazání). Tyto operace jsou prováděny v rámci tzv. transakcí (viz sekci 3.1.4). Pro specifikaci těchto operací nad konkrétními daty se v relačních databázích používá primárně jazyk SQL (*Structured Query Language*), ten umožňuje uživatelům vyhledávat data a manipulovat s nimi za pomoci popisu činnosti namísto programování konkrétních operací pro databázový systém. [31]

3.1.4 Transakční zpracování

Zpracování operací pomocí transakcí umožňuje bezkonfliktní spolupráci více uživatelů v jednom databázovém systému. Transakce je jedna celá akce (např. převod peněz z účtu na účet) a operace (z CRUD kategorií) jsou jednotlivé podčásti, které musí být provedeny, aby byla provedena celá transakce. Pro příklad převod peněz z účtu na účet se v řeči CRUD operací skládá ze zjištění, zda je na účtu dostatek peněz, snížení zůstatku na účtu odesílatele, zvýšení zůstatku účtu příjemce a záznamu o pohybu do výpisu z účtu.

V mnoha databázových systémech musí provedení transakce za všech okolností splňovat 4 základní podmínky, které se označují anglickou zkratkou ACID a jde o *atomicitu*, *konzistenci*, *integritu* a *trvanlivost*. Atomicita (**A**tomicity) znamená, že transakce musí být buď dokončena kompletně celá nebo nesmí být dokončena vůbec. Při jakémkoliv problému (např. nedostatku peněz na účtu) se veškeré operace od počátku transakce vrátí do původního stavu a transakce je ukončena. Konzistence (**C**onsistency) znamená, že během vykonávání transakce mohou být integritní omezení porušena, na konci transakce musí být všechna omezení splněna, tedy transakce musí databázi uvézt z jednoho konzistentního stavu do jiného, jinak není provedena. Izolace (**I**solation) je koncept, který požaduje, aby transakce prováděné souběžně v jednu chvíli měly stejný výsledek, jako kdyby byly prováděny postupně, tedy aby se vzájemně nemohly ovlivňovat. Trvanlivost (**D**urability) znamená, že databáze musí být stále v platném stavu a tento stav může být změněn pouze dokončenou transakcí, nikoliv softwarovou chybou, pádem apod.

Izolaci nebo serializovatelnost lze zajistit analýzou transakcí, které mají být provedeny souběžně a nalezením případných konfliktů, které by mohly vézt k odlišnému stavu databáze při postupném a souběžném provedení. Rizika konfliktů jsou řešitelná *pesimistickými* a *optimistickými* metodami. Pesimistické metody nedovolí souběh rizikových transakcí vůbec a pro zabezpečení proti konfliktům používají systém zámků, které transakci zaručí, že po dobu, kdy bude určitý objekt (např. tabulka) uzamčena, pak s ní nemůže pracovat žádná jiné transakce a namísto práce čekají ve frontě na uvolnění zámku. Optimistické metody předpokládají, že konflikty jsou vzácné a místo zámků a čekání ověřují výsledek transakce před jejím dokončením, čímž se snaží oproti pesimistickým metodám ušetřit čas. Proces transakcí je rozdělen na fáze čtení, ověření a zápisu, transakce si nejprve načte všechna data do odděleného pracovního prostoru, provede v něm změny a ověří, zda by aplikováním těchto změn neohrozila průběh jiné transakce a pokud ne, jsou změny z pracovního prostoru přepsány do samotné databáze. [31]

3.1.5 Příklady relačních databázových systémů

PostgreSQL

PostgreSQL je open source objektově-relační databázový systém, vyvinutý na University of California z projektu POSTGRES, jehož kořeny sahají až do roku 1986. Oproti ostatním relačním databázovým systémům je význačná dlouhodobá snaha o kompletní podporu a soulad se standardem SQL (aktuálně SQL:2016), široká uživatelská rozšiřitelnost, zahrnující především možnost uživatelsky definovat datové typy a možnost tvořit uživatelsky definované funkce v různých tzv. procedurálních jazycích. U PostgreSQL lze funkce programovat mimo jazyk SQL i např. ve variantách jazyků Perl a Python. V kontextu sekce 3.2 stojí za zmínku i podpora principu dědičnosti na úrovni databázových tabulek. [42]

MySQL a MariaDB

MySQL je nejpopulárnější open source relační databázový systém, je vyvíjen firmou Oracle Corporation [[37]], *MariaDB* je fork systému MySQL, vytvořený původními autory, aby i po odkupu MySQL firmou Oracle zůstal tento systém svobodný. Vývojáři MariaDB cílí na udržování vysoké míry kompatibility a funkční parity s MySQL a jeho API, aby MariaDB mohla sloužit jako přímá náhrada MySQL. [30]

SQLite

SQLite je open source databázový systém, který se od běžných databázových systémů liší především tím, že je nezávislý, nepotřebuje k provozu žádný server ani úvodní konfiguraci. Celá databáze je obsažena v podobě jediného přenositelného souboru na disku, přičemž se jedná o plnohodnotnou implementaci standardu SQL. Jedná se tak spíše o náhradu za ukládání aplikačních dat do souborů než o náhradu za výše zmíněné databázové systémy. [20]

QuestDB

QuestDB je open source relační databáze určená a optimalizovaná především pro data organizovaná v časových řadách. To pro uživatele znamená především možnost označit sloupec v tabulce jako časovou značku a podle těchto značek pak data spojovat, vybírat, agregovat a zpracovávat. Na druhé straně tato optimalizace zde znamená, vedle dalších drobných ústupků (ve srovnání s PostgreSQL absence části datových typů a části pokročilé funkcionality), především nemožnost upravovat či mazat již zapsané záznamy. Pro manipulaci s daty používá využívá jazyk ANSI SQL upravený a rozšířený pro lepší práci s časovými řadami. Pro připojení nabízí rozhraní kompatibilní s protokolem databází PostgreSQL a InfluxDB, případně umožňuje importovat či exportovat data a zasílat SQL dotazy pomocí REST API nebo integrovaného webového klienta. [39]

3.2 Objektově relační mapování (ORM)

Popularita relačního paradigmatu při práci s databázemi a rozšířenost objektově orientovaného paradigmatu mezi mnoha programovacími jazyky znamená, že se v rámci softwarových projektů tato paradigmata jistě potkají. Jejich principiální rozdíly vytváří vzájemný nesoulad, který musí programátoři takových aplikací řešit. Metody řešení těchto nesouladů se souhrnně nazývají jako *objektově relační mapování* (zkr. *ORM* z angl. *Object-Relational Mapping*). [26]

Principy relačního paradigmatu popisuje podsekce 3.1.2, je ale nutné si přiblížit ještě základní principy objektově orientovaného paradigmatu. Podle knihy [14] jsou objekty nezávislé entity, které mají identitu (lze je vzájemně identifikovat) a které jsou složeny z dat a operací (resp. atributů a metod). Operace obvykle slouží k práci a manipulaci s vnitřními daty objektu. Objekty taktéž mají mimo jiné vlastnosti zapouzdření a dědičnosti. *Zapouzdření* znamená, že objekty navenek skrývají svá data a stav a lze s nimi pouze komunikovat zasíláním zpráv, na které objekty reagují. Zasilatel nemá možnost dovnitř objektu nahlédnout nebo zasáhnout zvenčí. *Dědičnost* značí, že je možné definovat objekty na základě jiných objektů. Objekt, který je dědičností vytvoření na základě jiného přebírá všechny jeho atributy a metody a k nim může přidat další, případně těm existujícím může změnit chování.

Mnoho objektově orientovaných programovacích jazyků navíc využívá koncept tříd a instancí. *Třídy* jsou předem definovanou šablonou, která definuje atributy (data) a metody (operace). Tuto šablonu je pak možné instanciovat – vytvořit na jejím základě *instance* (objekty), které budou sdílet datové položky a operace definované třídou. Pro třídy taktéž platí vlastnosti popsané výše. [14]

Systémy ORM tedy umožňují programátorům využívat objektově orientované jazyky a přemýšlet nad datovým modelem aplikace v kontextu entit a jejich vztahů namísto v kon-

textu databázového schematu. Na pozadí tyto systémy monitorují změny v objektech a automaticky generují a provádí potřebné SQL dotazy realizující CRUD operace, které těmto změnám odpovídají. [36]

3.2.1 Příklady nástrojů pro objektově relační mapování

TypeORM

TypeORM je nástroj pro objektově-relační mapování pro jazyky JavaScript a TypeScript použitelný například na platformách Node.js, Ionic nebo Electron. Na straně databáze dokáže TypeORM pracovat mimo PostgreSQL i s řadou dalších databázových systému, např. MySQL, MariaDB, SQLite, MS-SQL i s nerelační dokumentovou databází MongoDB. Ve vztahu k databázi podporuje jak vysokoúrovňovou práci s entitami v podobě objektů, tak tvorbu dotazů, jejichž výsledky jsou následně převedeny na objekty, podobně jako knihovna SQLAlchemy popsaná dále. [43]

SQLAlchemy

SQLAlchemy modul pro jazyk Python, zahrnující sadu nástrojů pro práci s databázemi (*SQLAlchemy Core*) a nástroj pro objektově-relační mapování (*SQLAlchemy ORM*). SQLAlchemy přistupuje k nesouladům mezi relačním a objektovým modelem způsobem, který sice nabízí abstrakci a automatizaci práce s databází, ale stále nechává návrh relačního modelu, tabulek a vazeb mezi nimi primárně na vývojáři aplikace a do podoby tohoto modelu zasahuje pouze minimálně. K tomu dopomáhá zmíněné rozdělení na Core a ORM součást, kde lze podle potřeby kombinovat prvky ORM součásti s vysokou mírou abstrakce, tedy práce na úrovni entit a jejich vztahů nebo získat přímou kontrolu nad databází pomocí Core součásti, provádět konkrétní SQL dotazy a získávat jejich výsledky převedeny do formy objektů. [12]

3.3 Nerelační databáze

Relační databáze jsou stále standardem pro většinu databázových aplikací, ale začínají se objevovat různé aplikace, které naráží na požadavky, které SQL databáze splnit nedokáží. Příkladem může být potřeba zpracovávat velké objemy dat, například u velkých webových aplikací nebo zmíněných IoT systémech typu chytrých měst. SQL databáze jsou navrženy, jak již bylo v sekci 3.1 rozvedeno, s ohledem na integritu a korektní transakční zpracování, což je nutné například pro bankovní aplikace, nicméně u méně kritických dat tyto databáze velmi rychle dosáhnou svých limitů, protože zajišťování bezpečnosti a konzistence je náročné na výkon. [31]

3.3.1 Vlastnosti nerelačních databází

Podle knihy [31] mají nerelační (NoSQL) databáze jiný model než relační, především se podle ní jedná pak o *úložiště párů klíč-hodnota* (key-value store), *dokumentová úložiště* (document store), *grafové databáze* (graph database) a *databáze rodin sloupců* (column family database). Dále je kladen důraz na horizontální škálovatelnost, data nejsou omezena schématem nebo jen slabě, data lze snadno replikovat, lze k nim snadno přistupovat skrze nějaké API a konzistence databáze je zajištěna jiným modelem než modelem ACID (viz sekci 3.1.4), které umožňují různým uzlům, které dohromady tvoří škálovaný databázový systém

držet data v různých verzích a stavech, které se stanou vzájemně konzistentní v průběhu času.

3.3.2 Typy nerelačních databází

Úložiště párů klíč-hodnota

Tyto databáze způsobem ukládání dat evokují běžné programovací jazyky a jejich asociativní pole. Do nich je možné uložit nějaký datový objekt a označit jej klíčem, tedy jiným datovým objektem, podle kterého lze konkrétně právě jeden tento uložený objekt znovu dohledat. Tyto databáze nemají žádné schéma, což umožňuje velmi rychlé a snadné vkládání, vyhledávání a dělení této databáze mezi různé uzly, strukturovat klíče můžeme pouze manuálně, například použitím řetězců jako klíčů a v rámci řetězce jednotlivé logické části oddělit speciálním znakem.

Databáze rodin sloupců

Tento koncept rozšiřuje předcházející koncept párů klíčů a hodnot pomocí možnosti určitým způsobem tato data strukturovat. Využívá praktické zkušenosti z relačních databází, kde při načítání řádku z tabulky jsou zřídkakdy potřeba všechny sloupce, ale obvykle jsou při různých operacích potřeba různé skupiny (tedy *rodiny*) sloupců.

Výsledkem je databáze, která ukládá data do vícerozměrných tabulek adresovaných pomocí řádkových klíčů, vlastnosti objektů jsou adresovány taktéž pomocí sloupcových klíčů, sloupce jsou seskupeny do rodin sloupců, schéma tabulek popisuje pouze tyto rodiny a při distribuovaném nasazení se rodiny sloupců udržují pohromadě. Těmito vlastnostmi přináší výhodu škálovatelnosti úložišť párů klíč-hodnota, ale zároveň přinášejí možnost data strukturovat.

Dokumentová úložiště

Dokumentová úložiště ukládají data v podobě záznamů, zvaných *dokumenty* a byla vyvinuta především pro použití v rámci webových služeb, především sociálních sítí, vyhledávačů a kdekoliv, kde je požadavek na zpracovávání velkého množství heterogenních dat. Dokumentové úložiště nemá žádné schema pro data, což na jedné straně přenáší zodpovědnost za konzistenci dokumentů na uživatele a aplikaci, na druhé straně urychluje zpracování a umožňuje do databáze flexibilně ukládat různá data. Samotné dokumenty vnitřní strukturu mají, jsou identifikovány pomocí klíčů a uvnitř jsou strukturovány pomocí jmen atributů (sloužících jako klíče) a hodnot těchto atributů, jedná se tedy o variaci a rozšíření principů úložiště párů klíč-hodnota.

Grafové databáze

Grafové databáze se od ostatních typů liší zásadněji, data jsou uložena v grafové struktuře v podobě otypovaných uzlů a hran, kde uzly reprezentují objekty a hrany vztahy mezi nimi. Uzly i hrany jsou vždy pojmenované a mohou mít atributy, do kterých se ukládají samotná data. Schema této databáze je implicitní, není nutné tedy předem definovat jednotlivé typy hran a uzlů, místo toho jsou typy vytvořeny databázovým systémem automaticky až po vložení uzlu nebo hrany do databáze. K manipulaci s daty se využívají grafové operace využívající vlastnosti grafů jako jsou podgrafy, cesty grafem, apod. Využitím pro tyto da-

tabáze jsou případy, kde jsou propojení mezi daty důležitější než samotné datové záznamy – při analýze infrastruktur, pro zpracování dat sociálních sítí, atd.

3.3.3 Konkrétní příklady nerelačních databází

InfluxDB

InfluxDB je opensource nerelační databáze optimalizovaná pro data organizovaná v podobě časové řady. Způsob práce této databáze se od běžných relačních databází a některých typů nerelačních databází, kde jsou data identifikována klíčem, liší tím, že namísto klíče jsou data vázána k určité unikátní, vzestupně řazené časové značce v určité časové řadě. Touto vazbou je mimo jiné vyřešena i deduplikace dat – data, byť různá, vztahující se ke shodnému časovému okamžiku jsou považována za duplikáty a jsou obvykle zahozena. Databáze předpokládá, že jsou data v průběhu času pouze vkládána, nikoliv následně modifikována. Operace modifikace a mazání jsou tedy za účelem zvýšení výkonu omezeny. Dále pro zvýšení výkonu InfluxDB upřednostňuje rychlost požadavků před konzistencí, výsledky dotazů, které databáze vyřídí během probíhající transakce, tedy nemusí obsahovat nejnovější data. Podobně jako u mnohých jiných nerelačních databází, schéma ukládaných dat se specifikuje až při jejich vkládání, nikoliv předem, především proto, že data v časových řadách jsou obvykle krátkodobě ukládaná a jsou poměrně různorodá. Schéma uložené hodnoty se skládá z názvu měření, ke kterému se data vztahují, indexovaných tagů a neindexovaných pojmenovaných polí s jejich hodnotami. [24]

InfluxDB umožňuje nejen data ukládat a vybírat z databáze, ale umožňuje i zpracovávání, filtrací a agregaci dat během dotazování. K manipulaci s daty využívá vlastní funkcionální skriptovací jazyk Flux, jehož cílem je sjednotit vybírání, zpracovávání a analyzování dat do jednoho jazyka a jedné syntaxe. Dokumentace InfluxDB zpracování dat pomocí tohoto jazyka popisuje v analogii k úpravě vody, kdy je voda čerpána, projde několika stanovišti, které ji upravují a čistí až dojde k zákazníkovi, jazyk Flux řetězí podobným způsobem datové filtry až do bodu, kdy se k aplikaci dostanou požadovaná data v požadované formě. [23]

Amazon DynamoDB

Jedná se o cloudové řešení NoSQL databáze, které patří do skupiny služeb AWS firmy Amazon. Dynamo DB nabízí podporu ukládání párů klíč-hodnota i dokumentů a je automaticky škálovatelná jak do velikosti, tak počtu požadavků na zápis a čtení, tak replikace mezi regiony po světě. Nabízí integraci se službou AWS Lambda (viz podsekci 2.2.3), která umožňuje v této službě spouštět funkce například na základě změn v databázi. DynamoDB dále podporuje ACID transakce, šifrování, verzování databáze a automatické i ruční zálohování. [7]

3.4 Cloudové databázové služby

Provozovatelé cloudových platforem, jako jsou například Amazon, Microsoft nebo Google, poskytují služby typu SaaS (viz podsekci 2.2.1) zaměřené na provoz databázových serverů v cloudové infrastruktuře (viz sekci 2.2).

Podle produktových webů těchto provozovatelů (Amazon [5], Google [15] a Azure [34]) nabízí všichni podobné nebo srovnatelné služby. V nabídce Amazonu a Microsoftu se nachází vlastní proprietární relační databázový server, nabízející kompatibilitu minimálně

s rozhraním MySQL a PostgreSQL. Provozovatelé jako výhody tohoto řešení uvádí především optimalizaci jejich systémů pro nasazení v cloudovém prostředí, automatizované nasazení, zálohování a aktualizace, automatické škálování podle potřeb zákazníka, vysokou dostupnost a vysoké zabezpečení. Amazon navíc u své služby Aurora uvádí, že je násobně výkonnější než MySQL a PostgreSQL. Google nabízí pouze provoz systémů MySQL, PostgreSQL a Microsoft SQL Server v jeho cloudové infrastruktuře, obdobnou službu nabízí vedle proprietárního serveru i ostatní provozovatelé. I u tohoto typu služby zůstávají výhody dynamického škálování, vysoké dostupnosti apod.

Nabídka nerelačních databází se u všech tří velkých provozovatelů taktéž poměrně prolíná. Každý z poskytovatelů nabízí alespoň úložiště typu klíč-hodnota a dokumentovou databázi, obojí na proprietárním řešení. Amazon navíc nabízí i grafovou databázi a nerelační databázi pro časové řady. Microsoft naproti tomu různé typy nerelačních databází sdružuje do jedné vícemodelové služby s řadou API pro různé způsoby přístupu.

3.4.1 Amazon AWS

Jak zmiňuje úvod této sekce, platforma Amazon AWS nabízí provoz řady databázových systémů v cloudové infrastruktuře. Pro provoz relačních databází nabízí službu Amazon RDS, pro provoz nerelačních databází pak řadu konkrétních služeb podle typu nerelační databáze. Například dokumentovou databázi DocumentDB, grafovou databázi Neptune, úložiště klíč-hodnota DynamoDB nebo nerelační databázi pro časové řady Timestream. [5]

Amazon RDS zajišťuje nasazení, provoz, škálování a řízení databázových systémů v cloudu. Podporuje běžné databázové systémy, např. PostgreSQL, MySQL, MariaDB popsané v podsekci 3.1.5, ale i mnohé další. Mezi těmito systémy nabízí i vlastní databázový systém Amazon Aurora, kompatibilní s PostgreSQL i MySQL, Amazon u ní slibuje násobně vyšší výkon oproti variantám RDS s PostgreSQL nebo MySQL servery a lepší optimalizaci pro fungování v cloudové infrastruktuře. [4]

Rozhraní pro IoT zařízení

Jak popisuje sekce 2.3 o technologiích IoT, prakticky standardem pro komunikaci IoT zařízení s obslužnými servery, případně cloudovou platformou je protokol MQTT. Pro propojení těchto zařízení komunikujících protokolem MQTT, případně HTTPS nebo LoRaWAN[1] s databází a dalšími službami v cloudu nabízí Amazon v rámci platformy AWS službu IoT Core.

AWS IoT Core vytváří zabezpečené rozhraní pro spojení a komunikaci IoT zařízení s cloudovými službami. Zprávy, které jsou na tomto rozhraní přijaté, lze filtrovat, transformovat a spouštět na jejich základě další služby v závislosti na vyhodnocení definovaných pravidel. Lze tedy z přijaté zprávy uložit požadovaná data do databáze, případně s jejich pomocí spustit požadovanou funkci služby AWS Lambda (viz podsekci 2.2.3), která data může dále zpracovat a uložit. IoT Core umožňuje obousměrnou komunikaci, lze skrze něj tedy nejen přijímat data ze zařízení, ale i posílat zařízením příkazy. [2]

3.4.2 Manuální nasazení

Zmíněné databázové systémy jako například MariaDB, PostgreSQL, InfluxDB apod. lze provozovat i mimo cloudové platformy. Pro zajištění škálování a replikace dat existují

[1] Energeticky úsporný protokol určený k propojení IoT zařízení na velké vzdálenosti s bránami, které poskytují přístup do internetu [13]

pro tyto systémy různé nástroje nebo modifikace, pro MariaDB a MySQL například Galera Cluster [29], pro PostgreSQL například Postgres-XL [41], InfluxDB má pro tento účel placenou edici Enterprise [25].

Postgres-XL vychází z PostgreSQL a umožňuje škálovat databázi jak pro zápis, tak pro paralelní zpracovávání dat na více instancích serveru propojených do tzv. clusteru. Z uživatelského pohledu je toto rozšíření transparentní, rozhraní je tedy shodné s PostgreSQL a lze s ním pracovat stejným způsobem jako s instancí PostgreSQL na jednom serveru bez ohledu na to, jakým způsobem je cluster nasazen a organizován.

Cluster Postgres-XL se skládá z 3 komponent: *transakčního manažeru*, který se stará o konzistenci transakcí, *koordinátora*, který poskytuje aplikační rozhraní clusteru, tedy přijímá SQL dotazy a řídí jejich provedení a samotných *datových uzlů*, které obsahují uložená data, přijímají zprávy od koordinátora a podle manipulují s daty.

Škálování pro zápis znamená, že s rostoucím počtem serverů v clusteru roste počet požadavků na zápis, které je schopen server vyřídit, přičemž jsou tato data ihned zviditelněna i pro ostatní servery. Data jsou při zápisu rozložena na různé servery v rámci clusteru a v závislosti na nastavení případně i zreplikována. *Paralelní zpracování dat* u Postgres-XL znamená, že koordinátor sestaví plán vykonání transakce, vyhodnotí, které datové uzly se vykonání budou účastnit a poté jednotlivým uzlům předá instrukce, které uzly paralelně vykonají a výsledek předají zpět koordinátorovi. [41]

Kapitola 4

Analýza

Tato kapitola uvádí v sekci 4.1 základní popis systému pro chytrá města firmy Logimic spolu s bližší specifikací řešeného problému. Dále v sekci 4.2 popisuje požadavky na aplikaci. V závěru kapitoly sekce 4.3 analyzuje existující nástroje a porovnává je s požadavky firmy Logimic.

4.1 Systém pro správu chytrých měst

Firma Logimic[1], v jejíž spolupráci tato práce vzniká, se zabývá vývojem, nasazením a provozem řešení pro chytrá města. Jedná se o systém zahrnující IoT zařízení, (např. chytré pouliční osvětlení, teploměry, měřiče vodní hladiny, řízení vytápění, detektory kouře, apod.), cloudový systém pro sběr a zpracování dat a uživatelské rozhraní pro monitoring a řízení těchto zařízení v několika úrovních abstrakce.

4.1.1 Aktuální situace

V současné době je systém ve vývoji a testování probíhá v rámci limitovaného nasazení na omezeném počtu zařízení, s omezeným počtem uživatelů přistupujících do systému. Očekává se, že v době nasazení systému do ostrého provozu bude systém pracovat s řádově vyšším množstvím IoT zařízení ve více městech a že bude se systémem interagovat řádově více uživatelů.

Při vývoji a takto omezeném nasazení a testování není firmě jasné, jak se bude systém chovat při ostrém nasazení. V první fázi se očekává, že bude součástí systému řádově tisíc zařízení a desítky uživatelů denně. Postupem času by měl počet zařízení i uživatelů dále narůstat. Podobně nejsou jasné výkonnostní požadavky na hardware s ohledem na množství uživatelů a finanční náklady na provoz systému na cloudové platformě.

Z interní dokumentace a konzultací vyplynulo, že systém má při provozu 3 hlavní zdroje, které se významnou měrou podílí na jeho zátěži. Prvním jsou zmíněné příchozí zprávy z IoT zařízení, které jsou zpracovány a zapsány do databáze. Druhým je rozhraní pro frontend aplikace, které zpracovává a poskytuje data frontendové části systému na základě požadavků uživatelů. Posledním je periodicky spouštěný výpočet, který zpracovává data a parametry z IoT zařízení do tzv. KPIs *(Key Performance Indicators)*[2].

[1] Viz https://www.logimic.com

[2] Jako KPIs interně označujeme spočtené metriky na různých úrovních abstrakce nad IoT zařízeními v systému

4.1.2 Technologie systému

V současné době je systém prakticky kompletně provozován v rámci cloudové platformy AWS firmy Amazon. Jednotlivé operace v systému (např. vytvoření uživatele, zápis hodnot ze zařízení, výčet zařízení pro frontend, apod.) jsou rozděleny do funkcí pro platformu AWS Lambda (viz podsekci 2.2.3). Data z IoT zařízení jsou přijímána skrze protokol MQTT pomocí služby AWS IoT Core, která na základě přijatých zpráv spouští obslužné funkce v AWS Lambda. Pro ukládání dat se využívá relační databázový systém PostgreSQL, nasazený v rámci služby Amazon RDS (viz podsekci 3.4.1). Dále se naměřená data ze senzorů shromažďují v časové řadě v nerelační databázi InfluxDB (viz podsekci 3.3.3).

Možnost nasadit systém pro testování i provoz mimo cloudové prostředí je ve vývoji, ale funkcionalita dostupná prostřednictvím funkcí pro AWS Lambda je implementovaná i pro běhové prostředí Node.js, ale chybí vhodné rozhraní pro využití této funkcionality, které jinak poskytuje AWS IoT Core a další služby cloudové platformy AWS.

4.2 Požadavky na řešení

Na základě aktuálního stavu vyplývá, že hlavním problémem je nemožnost odhadnout, jak se systém bude chovat po nasazení do ostrého provozu a tím vzniká požadavek na nalezení způsobu testování systému bez účasti reálných IoT zařízení a uživatelů. Z aktuálního stavu a úvodních konzultací se zástupcem firmy postupně vyplynul požadavek na vytvoření testovacího softwaru na míru architektuře systému a sada požadavků, jak by testování mělo fungovat a probíhat.

4.2.1 Testovací nástroj

Testovací software má testovat především chování relační databáze a dle jejího stavu i chování zbytku systému na základě úrovně softwarem generované zátěže. Generovaní zátěže se má soustředit na 3 hlavní zdroje zátěže zmíněné v sekci 4.1.1. Tedy napodobovat chování IoT zařízení a generovat zprávy protokolu MQTT odpovídající obsahem skutečným zařízením, požadovat různá data určená pro zobrazování na frontendu a periodicky spouštět přepočty hodnot KPI. Generátory musí být uživatelsky konfigurovatelné, odděleně podle zdroje zátěže. Navíc tyto generátory musí být oddělitelné a spustitelné v případě potřeby v několika různých lokalitách souběžně při zachování možností je řídit a sbírat z nich data.

Dále mají být shromažďována a ukládána relevantní získatelná data o stavu a vytížení databáze spolu s informacemi o množství zasílaných požadavků z jednotlivých kategorií a monitorování případné fronty požadavků. V testovacím softwaru musí být možnost zatěžování monitorovat a sesbíraná data z testování vhodně vizualizovat, aby bylo možné data po skončení testu vyhodnotit a zkoumat chování databáze v závislosti na míře vytížení.

4.2.2 Úprava rozhraní

Během následných konzultací vyvstal navíc požadavek na vytvoření jednoduché brány pro přístup k API firmy Logimic, kterou lze spustit v lokálním prostředí na platformě Node.js. V této konfiguraci má být možné tuto bránu využít v kombinaci se zmíněným testovacím nástrojem pro měření výkonu mimo cloudovou platformu na různém hardwaru a přibližně určit hardwarovou náročnost tohoto řešení pro různé úrovně zátěže. Tato brána by také měla umožnit vykonávání více požadavků souběžně. Tím by mělo být eliminováno případné úzké hrdlo v podobě jednovláknového zpracování požadavků a přiblíží se tím

reálnému rozhraní. To je automaticky škálované a může nastat situace, kdy se vykonává několik požadavků souběžně.

4.2.3 Shrnutí hlavních požadavků

- Nástroj má produkovat zátěž odpovídající třem hlavním zdrojům zátěže v reálném nasazení.
- Tato zátěž má být konfigurovatelná, aby bylo možné odsimulovat různé úrovně zatížení.
- Zdroje zátěže má být možné rozdělit do více oddělených lokalit.
- Nástroj má shromažďovat a zaznamenávat data o generování zátěže a vytížení databáze.
- Nástroj má nabízet možnost, jak sesbíraná data vhodně vizualizovat.
- Rozhraní firmy Logimic je nutné rozšířit o bránu pro lokální použití.

4.3 Existující nástroje

Následující podsekce obsahují stručný přehled nástrojů pro testování zátěže, jejich funkcí a vlastností. Závěrečná podsekce 4.3.4 obsahuje zhodnocení vhodnosti těchto nástrojů vůči požadavkům firmy Logimic.

4.3.1 Artillery

Jedná se o testovací nástroj pro backend schopný zátěžově testovat webová rozhraní nebo API na různých protokolech, např. HTTP, WebSocket nebo Socket.io, za pomocí pluginů pak umožňuje testování např. pomocí SQL dotazů nebo spouštěním funkcí platformy AWS Lambda. Umožňuje použití vlastních skriptů, které mohou ovlivňovat běh testování, starat se o vykonávání určitých operací na testovaném API nebo sbírat a ukládat vlastní metriky, které není schopen nástroj uložit sám o sobě. Nasbírané metriky však Artillery neumí zobrazovat a interpretovat, umí je pouze za pomoci pluginu předat jiné monitorovací aplikaci, jako je Prometheus nebo Datadog. [10] Komunitní verze tohoto nástroje je zdarma a opensource, řešení pro spouštění testů z platformy Amazon AWS je v omezené verzi taktéž zdarma, placená verze umožňuje detailnější nastavení a spouštění více souběžných testů, přičemž cena se pohybuje od 1 119 do 3 990 dolarů měsíčně. [11]

4.3.2 Apache JMeter

Je aplikace pro zátěžové testy a měření výkonu. Je schopna testování a měření na řadě protokolů, např. HTTPS, FTP, nebo SMTP a IMAP. Umožňuje pomocí konektoru JDBC připojení a testování databázových systémů, případně umožňuje spouštět na cílových stanicích nativní příkazy a shellové skripty. Dokáže o průběhu testování vygenerovat zprávu v HTML. Podporuje taktéž distribuované spouštění testů. Informace o probíhajícím testování shromažďuje ale pouze z odpovědí na zaslané požadavky. [9]

4.3.3 Grafana k6

Grafana k6 umožňuje zátěžové testování pomocí testů napsaných v jazyce JavaScript. Umožňuje testovat především API s HTTP či WebSocket rozhraním. Podporu o další protokoly lze do k6 doplnit pomocí rozšíření v jazyce Go. O testech umožňuje podobně jako Artillery shromažďovat data do monitorovacích aplikací jako například Prometheus a Datadog, dále pak do databázových systémů jako například InfluxDB nebo do souborů formátu JSON a CSV. Po testu k6 generuje zprávu o výsledcích. [17] Open source verze k6 lze na vlastním hardwaru používat zdarma, placená verze nabízí kapacitu pro testování na cloudové platformě a vizualizaci výsledků testování, ceny se pohybují od 99 po 1499 dolarů měsíčně pro běžné uživatele a řešení na míru od 25 tisíc dolarů ročně pro velké společnosti. [18]

4.3.4 Zhodnocení

Pro zátěžové testování aplikačních rozhraní a databázových systému existuje několik řešení, nabízejících poměrně podobné služby. Žádné z popsaných řešení neplní specifikované požadavky zcela. Všechna popsaná řešení umožňují rozšiřovat svoji funkcionalitu a schopnosti za pomoci řešení.

Na základě dokumentace k nástroji Artillery lze předpokládat, že by bylo možné využít dostupná aplikační rozhraní pro vývoj rozšíření a doplnit schopnosti a funkce tohoto nástroje, aby požadavky splnil.

Zbylá dvě řešení jsou taktéž open source a mají podporu pro rozšíření, nicméně z jejich dokumentace vyplývá, že rozšířit je o potřebnou funkcionalitu by bylo v dostupném čase prakticky nemožné nebo citelně náročnější než u aplikace Artillery.

Kapitola 5

Návrh testovacího softwaru

Tato kapitola popisuje v sekci 5.1 navržené principy zátěžového testování rozhraní systému firmy Logimic. Dále sekce 5.2 popisuje návrh architektury a jednotlivých komponent testovacího nástroje.

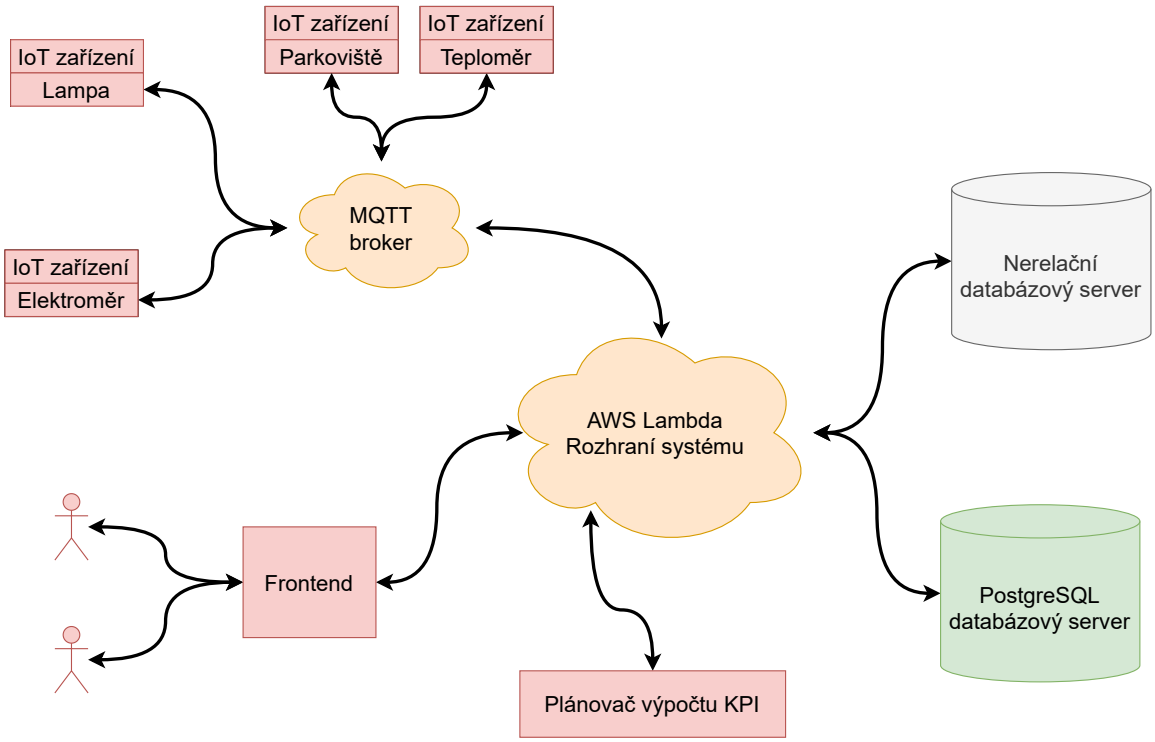
5.1 Princip testování

Podsekce o aktuálním stavu (4.1.1) systému firmy Logimic popisuje dominantní zdroje zátěže na tento systém. Na základě tohoto popisu byly identifikovány konkrétní operace poskytované rozhraním, které lze využít pro zatěžování systému. Z pohledu IoT zařízení se jedná o jedinou operaci, která je zodpovědná za zpracování a uložení zprávy, podobně periodickou zátěž tvoří pouze jediná operace, zodpovědná za přepočet KPIs. Z operací, které poskytují data frontendové části systému byla vybrána především operace získávající data o konkrétním KPI, dále pak operace získávající výčet zařízení s jejich stavem.

Spouštění těchto operací probíhá skrze rozhraní, na které jsou zasílány řídící zprávy s názvem operace a parametry pro její spuštění. Z pohledu testovacího nástroje jsou tyto zprávy zasílány množstvím běžících komponent – generátorů zátěže. Ty jsou zodpovědné vždy za jeden typ požadavku, generování jeho náhodného obsahu a načasování zasílání zpráv, aby byla dosažena nakonfigurovaná úroveň zátěže systému a byl splněn požadavek na konfigurovatelnost intenzity zátěže.

Komponenty, které svým provozem zatěžují rozhraní, je nutně nějak ovládat a provádět sběr a zpracování dat o provedených požadavcích. Navíc je nutné splnit požadavek na rozdělení zdrojů zátěže do několika lokalit. Proto budou skupiny zatěžovacích komponent mít své řídící komponenty, které budou zodpovědné za jejich spouštění, běh, řízení, ukončení a za průběžný sběr dat. Všechny řídící komponenty by poté měly být spravovány jednou centrální komponentou, která bude proces testování řídit, monitorovat, bude sbírat zpracovaná data z řídících komponent a ukládat je. Centrální komponenta by měla taktéž poskytovat rozhraní pro řízení testování uživatelem a pro poskytování informací, vizualizací a dat uživateli.

Obrázek 5.1 zjednodušené zobrazuje architekturu testovaného systému. Červeně jsou vyznačeny hlavní zdroje zátěže, které testovací nástroj nahrazuje a jejich činnost simuluje. Oranžově je vyznačeno rozhraní systému a brána k rozhraní, které je pro testování nutné rozšířit o zmíněnou možnost lokálního spuštění a monitorování. Zeleně je vyznačena databáze, jejíž chování má nástroj během generování zátěže monitorovat.



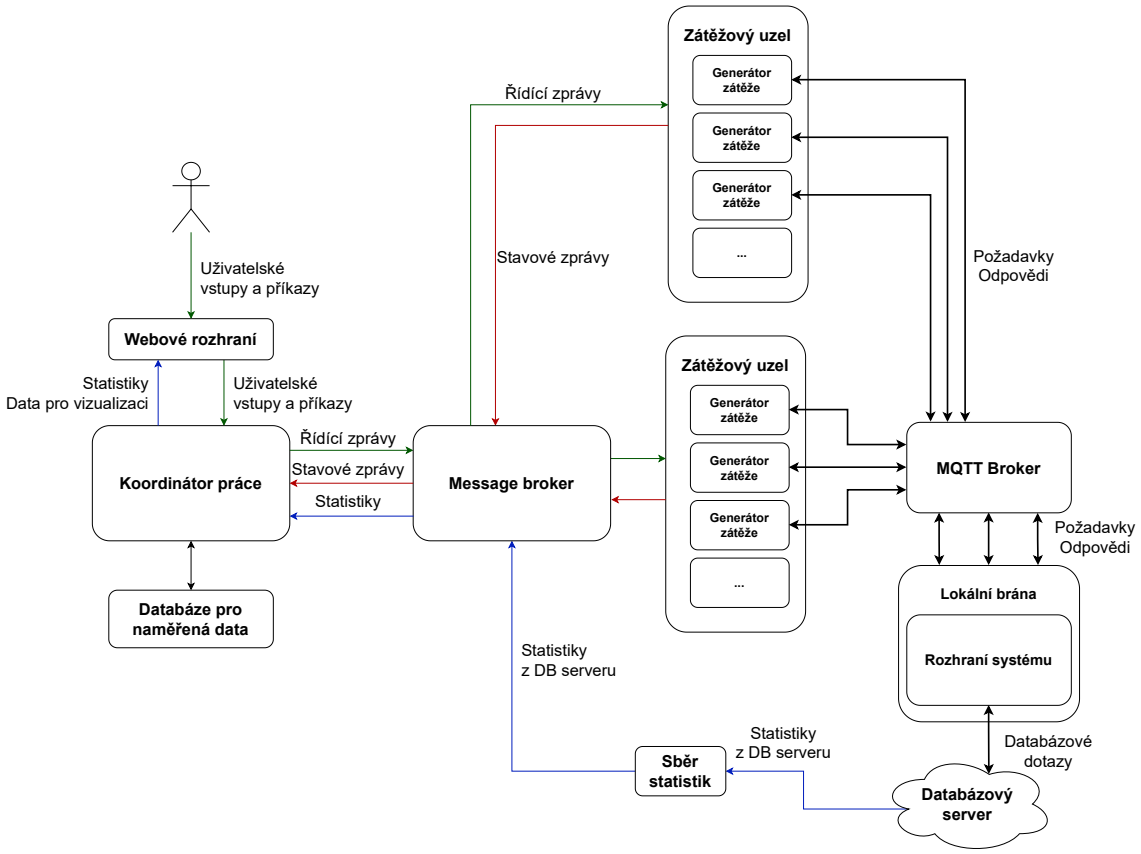
Obrázek 5.1: Architektura systému firmy Logimic

5.2 Architektura testovacího nástroje

Testovací nástroj je na základě principů zmíněných v předchozí sekci 5.1 navržen jako soubor vzájemně komunikujících a spolupracujících součástí. Jedná se o koordinátor práce a jeho uživatelské rozhraní (centrální komponenta systému) a zátěžové uzly (řídící komponenty s integrovanými generátory zátěže). Činnost hlavních dvou komponent doplňuje samostatná komponenta pro sběr statistik z databázového systému, databáze pro ukládání naměřených dat formou časové řady a message broker pro vzájemné propojení součástí.

Uživatelské rozhraní koordinátora práce je navrženo jako webová aplikace, ostatní komponenty jsou navrženy jako aplikace s textovým rozhraním.

Obrázek 5.2 zobrazuje architekturu testovacího nástroje společně s vazbami a komunikací mezi jednotlivými komponentami a testovaným rozhraním.



Obrázek 5.2: Náčrt architektury testovacího nástroje

5.2.1 Koordinátor práce

Tato součást řídí celý průběh testování, komunikuje se zátěžovými uzly, instruuje je k plnění uživatelem definovaných úloh a testovacích scénářů, přijímá z nich a ze služby pro sběr statistiky data, která zpracovává, agreguje a ukládá je do databáze. Udržuje a sbírá informace o stavu všech uzlů a poskytuje nasbíraná data pro generování výstupních grafů pro uživatele. Je úzce spojena s webovým serverem, který poskytuje uživatelské rozhraní pro řízení a sledování činnosti nástroje vč. vizualizace nasbíraných dat z testování.

Řízení průběhu testování se liší podle aktivního režimu, ty jsou dva – manuální a automatický podle předkonfigurovaného scénáře. V manuálním režimu koordinátor skrze webové rozhraní přijímá příkazy od uživatele a na jejich základě zasílá řídící zprávy dostupným zátěžovým uzlům. Lze spouštět komponenty pro generování zátěže, těm jednotlivě přidělovat úlohy (volit kterou oblast zátěže mají simulovat), tyto úlohy spouštět, zastavovat a měnit jim rychlost zasílání požadavků. V automatickém režimu se koordinátor práce postará o spuštění požadovaného množství generátorů zátěže na požadovaném množství zátěžových uzlů, přidělení úloh a nastavení rychlostí dle konfiguračního souboru. Navíc, pokud to konfigurační soubor specifikuje, může uzlům nastavit automatické zrychlování zasílání požadavků za účelem hledání bodu, kdy rozhraní přestane stíhat požadavky vyřizovat.

Sběr dat o testování probíhá příjmem zpráv ze zátěžových uzlů a komponenty pro sběr statistik skrze message broker (viz sekci 5.2.6). Data ze zpráv jsou následně zpracována, prezentována uživateli a pravidelně ukládána do databáze pro pozdější použití.

Data shromažďovaná koordinátorem práce

Pro vykonávání své činnosti potřebuje koordinátor práce uchovávat následující strukturu dat.

- informace o stavu jednotlivých zátěžových uzlů
  - ID
  - čas, kdy se naposled ohlásil koordinátorovi
  - úkoly, které je schopen vykonávat
  - seznam generátorů zátěže a informací o nich (seznam instancí třídy `WorkerMeta`)
    - ID
    - úkol, který pracovník vykonává (např. typ zátěže, kterou generuje)
    - stav
    - rychlost zasílání požadavků
    - počty požadavků
      - odeslaných celkem
      - odeslaných bez odpovědi
- informace o testovacím scénáři, pokud probíhá
- agregovaná statistická data
  - počty požadavků – odeslaných, nezpracovaných
  - počty detekovaných chyb
  - celková rychlost zasílání požadavků
  - počty procesů databázového serveru – celkem, aktivních, čekajících, čekajících v transakcích
  - počty dokončených a nedokončených databázových transakcí
  - počty n-tic, které databázový systém vybral, vrátil, smazal, zapsal a upravil

5.2.2 Uživatelské rozhraní

Webové uživatelské rozhraní je složeno ze tří záložek – manuálního řízení testování, řízení testování pomocí scénáře a vizualizace nasbíraných dat. Ve všech záložkách je zakomponována tabulka s aktuálními počty požadavků, databázových procesů, transakcí a řádků.

Záložka pro manuální řízení obsahuje lištu s ovládacími prvky, pomocí kterých lze manuálně spouštět a ukončovat instance pracovníků, přidělovat jim úkoly a nastavovat těmto úkolům rychlost zasílání požadavků. Dále umožňuje zaslat příkaz pro resetování nasbíraných dat o testování a příkaz k resetování celého testování ukončením všech generátorů zátěže.

Záložka pro testování pomocí scénáře nabízí výběr scénářů, možnost nahlédnout do nastavení scénáře a vybraný scénář spustit. Ukončení scénáře je možné z upozornění, které se po jeho spuštění začne objevovat nad statistikami.

Záložka s vizualizací nasbíraných dat nabízí čtveřici grafů rozdělených podle kategorií dat na graf počtů požadavků, počtů procesů databázového serveru, počtů transakcí a počtů

řádků. Ve všech grafech je navíc vynesena rychlost zasílání požadavků, aby bylo možné naměřené hodnoty srovnat s úrovní zátěže. Grafy se automaticky obnovují a vizualizují aktuální průběh testování nebo lze vybrat určitý časový úsek, pro který jsou grafy zobrazeny.

Náčrty jednotlivých panelů uživatelského rozhraní se nachází v příloze A.

5.2.3 Zátěžové uzly

Každý zátěžový uzel na základě řídících zpráv od koordinátora práce spouští, ukončuje a spravuje generátory zátěže. Těm následně může být přiděleno vykonávání úlohy, typicky generování zátěže odpovídající jednomu za tří hlavních typů. Data o těchto generátorech, jejich stavu, úkolech a statistikách uzel pravidelně zasílá koordinátorovi práce.

Generátor zátěže reprezentuje jednotku, která generuje náhodná data ve formátech určených předpisem úlohy, tato data zasílá ve formě požadavku na testované rozhraní resp. jeho bránu a udržuje přehled o těchto požadavcích. Dále generátory obsahují plánovač, pomocí kterého časují zasílání požadavků, aby dosáhly požadované intenzity zátěže. Generátory jsou spouštěny, ukončovány a řízeny přímo zátěžovými uzly, kterým generátor poskytuje rozhraní pro získávání dat o své činnosti a o zasílaných požadavcích.

Úkoly pro generátory zátěže reprezentují předpis, který zahrnuje výchozí rychlost zasílání požadavků, formát požadavku a způsob generování náhodných dat, kterými se vyplní datová pole požadavku.

5.2.4 Komponenta pro sběr statistik

Jedná se o skript, který se periodicky dotazuje databázového systému PostgreSQL, který je využíván testovaným rozhraním na hodnoty z kolektoru statistik[1]. Tato data jsou následně zformátována a zaslána koordinátorovi práce.

Hlavním důvodem pro vyčlenění této komponenty z koordinátora práce, jehož součástí by též mohla být, je umožnit spuštění této komponenty na stejném stroji nebo síti, kde je spuštěn i databázový server testovaného systému bez nutnosti povolit přístup k tomuto serveru i z Internetu.

5.2.5 Databáze pro sběr naměřených dat

Testovací nástroj sbírá data o průběhu testování z vlastních generátorů zátěže a z komponenty pro sběr statistik. Vlastní generátory poskytují informace o intenzitě zátěže, celkovém počtu zaslaných požadavků a počet aktuálně nedokončených požadavků. Komponenta pro sběr statistik poskytuje informace o počtech dokončených a zrušených transakcí, počty manipulací s řádky databáze a počty spuštěných procesů a jejich stav. Tyto informace jsou do databáze ukládány pravidelně na základě vzorkování koordinátoru práce a taktéž jsou pravidelně vybírány a zpracovávány do formy grafu těchto hodnot v čase.

5.2.6 Message broker

Úlohou message brokeru[2] v rámci tohoto nástroje je zprostředkování komunikace mezi koordinátorem, uzly a službou pro sběr statistik, které mohou být rozděleny na samostatných strojích v lokální síti nebo internetu. Komponentám pak stačí, aby se byly schopny

[1] https://www.postgresql.org/docs/current/monitoring-stats.html

[2] Pro osvětlení pojmů *message broker* a *téma* může posloužit podsekce 2.3.1 o protokolu MQTT

připojit na společný message broker. V rámci message brokeru využívají komponenty několik témat, která jsou blíže popsána v následujících odstavcích.

Uzly odebírají společné téma (`/backends`), které nyní slouží jen pro manuální požádání všech brokerů o zprávu o jejich stavu, kterou jinak zasílají periodicky bez vyzvání. Pro ovládání jednotlivých uzlů odebírají uzly témata ve formátu (`/backends/<unikátní ID uzlu>`, kde přijímají řídící zprávy od koordinátora práce, na jejichž základě ovládají generátory zátěže.

Brána testovaného rozhraní

Brána k testovanému rozhraní (viz sekci 5.3) využívá, jak již bylo zmíněno, MQTT broker k příjmu požadavků. Tento broker může dle implementace být shodný i oddělený od brokeru spojující komponenty.

Pro příjem zpráv využívá brána testovaného rozhraní téma `/lmbd`, samotné zprávy pro rozhraní jsou pro účely jejich sledování doplněny informacemi o generátoru, uzlu a pořadovém čísle. Po vyřízení požadavku generátory a brána testovaného rozhraní využívají podtémata v rámci tématu `/lgmcsim/response` – brána publikuje do podtématu ve formátu `/response/<unikátní ID uzlu>/<pořadové číslo generátoru>` zprávu s pořadovým číslem dokončeného požadavku.

Koordinátor práce využívá téma `/coordinator`, skrze které pro něj uzly a komponenta pro sběr statistik periodicky publikují data o svém stavu a stavu databáze.

5.3 Lokální brána k systému firmy Logimic

Systém firmy Logimic bylo za účelem testování v kontrolovatelném prostředí nutné rozšířit o součást umožňující běh tohoto rozhraní mimo prostředí platformy AWS Lambda a zároveň o pomocnou funkcionalitu usnadňující samotné testování.

Brána přijímá požadavky pomocí protokolu MQTT. Tyto požadavky jsou rovnoměrně rozdělovány mezi procesy, které jsou zodpovědné za vykonání operací, specifikovaných v požadavcích. Každý z procesů má frontu, do které příchozí požadavky řadí a postupně je z fronty odebírá a vykonává je pomocí volání funkcí rozhraní testovaného systému.

Po vykonání požadavku procesy komunikují skrze message broker s komponentami testovacího nástroje (viz podsekci 5.2.6), zasílají mu zprávu, obsahující pořadové číslo požadavku, který byl právě dokončen.

Kapitola 6

Implementace

Tato kapitola popisuje v sekci 6.1 výběr technologií, zvolených pro realizaci komponent popsaných v kapitole 5. Dále se v sekci 6.2 vrací k jednotlivým součástem testovacího nástroje a blíže představuje detaily jejich implementace.

6.1 Zvolené technologie

Podsekce 6.1.1 zahrnuje výběr a zdůvodnění jazyka pro tvorbu komponent testovacího nástroje obecně. Podsekce 6.1.2 přibližuje technologie a knihovny využité při tvorbě webového uživatelského rozhraní. Technologie využité k propojení komponent mezi sebou a komunikace s testovaným rozhraním rozebírá podsekce 6.1.3. Výběr databázového systému pro uchovávání dat z testování popisuje podsekce 6.1.4 a v závěru přibližuje zvolené technologie pro bránu k rozhraní systému podsekce 6.1.5. Koordinátor práce, zátěžové uzly a generátory zátěže zde explicitně zmíněny nejsou, protože výběr technologií pro ně je popsán v rámci podsekcí o zbylých komponentách.

6.1.1 Komponenty testovacího nástroje

Při volbě prostředí a jazyka pro implementaci jednotlivých komponent testovacího nástroje jsem se rozhodoval mezi jazykem TypeScript a běhovým prostředím Node.js a jazykem Python. Na základě mých výrazně větších zkušeností s jazykem Python 3 jsem pro implementaci zvolil jej.

6.1.2 Uživatelské rozhraní

Uživatelské rozhraní jsem se rozhodl implementovat v podobě webové aplikace. Hlavními důvody pro toto rozhodnutí byly výhody v podobě možnosti toto rozhraní využít i vzdáleně, zkušeností s implementací webových rozhraní a možnosti snadno pomocí existujících knihoven integrovat i vizualizační část nástroje do jednoho celku.

Jako webový framework pro jazyk Python jsem zvolil Flask[1]. Tento framework jsem zvolil primárně pro jeho jednoduchost a minimalističnost. Webové rozhraní se stará primárně o konfiguraci a monitoring ostatních komponent, k čemuž základní funkce nabízené frameworkem Flask dostačují a nadbytečné funkce nekomplikují implementaci a použití. Pro rozšíření jeho funkcionality jsou využita pouze dvě rozšíření – Flask-SocketIO[2] pro

[1] https://flask.palletsprojects.com/en/2.1.x/
[2] https://flask-socketio.readthedocs.io/en/latest/getting_started.html

komunikaci webové aplikace se serverem skrze knihovnu Socket.IO a Flask-MQTT[3] pro komunikaci s ostatními komponentami. Jako webový server pro modul Flask je kvůli podpoře protokolu WebSocket namísto vestavěného serveru použita knihovna gevent[4].

Frontend využívá framework Bootstrap 5[5], zvolil jsem jej díky jeho snadnému použití, rozsáhlé dokumentaci a mým zkušenostem s jeho použitím. Vizualizace nasbíraných dat o vytížení systému je realizována pomocí grafů, které jsou generovány ve webovém prohlížeči knihovnou Chart.js[6]. Mezi hlavní důvody pro volbu této knihovny patří především interaktivita výsledných grafů, která umožňuje uživateli lepší orientaci v datech, široké možnosti konfigurace grafů a kvalitní dokumentace knihovny.

Propojení frontendu s webovým serverem je realizováno pomocí JavaScriptových knihoven jQuery[7] a Socket.io[8] na frontendu a již zmíněného rozšíření Flask-SocketIO v rámci webového serveru. Knihovna jQuery je využita pro zjednodušení práce s asynchronními požadavky na webový server. Ty se využívají pro pravidelné aktualizace zobrazených komponent webového rozhraní. Knihovna Socket.io se využívá pro zasílání notifikací o změně dat mezi webovým serverem a frontendem.

6.1.3 Propojení komponent nástroje

V sekci 5.2.6 je propojení jednotlivých komponent specifikováno pomocí message brokeru. Při výběru konkrétního message brokeru jsem zvažoval použití brokeru RabbitMQ[9], úložiště Redis[10] a MQTT brokeru Eclipse Mosquitto[11] a případné využití dalších funkcí těchto aplikací v rámci jednotlivých komponent testovacího nástroje.

Ve finále jsem využil MQTT broker Eclipse Mosquitto. Hlavním důvodem pro jeho výběr bylo možnost při vývoji a testování využívat jednu běžící instanci brokeru jak pro komunikaci komponent mezi sebou, tak pro zasílání řídících zpráv bráně testovaného rozhraní. Dále pak tento výběr zjednodušil závislosti celého projektu, protože protokol MQTT by byl bez ohledu na propojení komponent využíván ke komunikaci s bránou testovaného rozhraní. Navíc v rámci návrhu nebyl specifikován požadavek na message broker, kvůli kterému by tento výběr vynutil kompromisy v jiných částech aplikace.

Ve všech komponentách implementovaných v jazyce Python je využita ke komunikaci knihovna Eclipse Paho MQTT[12].

6.1.4 Ukládání naměřených dat

S přihlédnutím k formátu naměřených dat (sada hodnot vzorkovaných v čase) jsem se rozhodl využít databázi optimalizovanou pro práci s časovými řadami. Konkrétně jsem zvolil databázi QuestDB[13] (více viz podsekci 3.1.5). Hlavním důvodem bylo její snadné nasazení a přenositelnost – jedná se o multiplatformní aplikaci v jazyce Java. Dalšími výhodami, které jsem při výběru zohlednil byl vestavěný webový klient, pomocí kterého lze procházet

[3] https://flask-mqtt.readthedocs.io/en/latest/
[4] http://www.gevent.org
[5] https://getbootstrap.com
[6] https://www.chartjs.org
[7] https://jquery.com
[8] https://socket.io
[9] https://www.rabbitmq.com
[10] https://redis.io
[11] https://www.mosquitto.org
[12] https://www.eclipse.org/paho/
[13] https://questdb.io

surová naměřená data a možnost zasílat požadavky bez použití klientské knihovny pomocí HTTP požadavků.

Za komunikaci s databázovým serverem je zodpovědný koordinátor práce, u kterého jsem pro jednoduchost zvolil právě HTTP rozhraní databázového systému a využívám ke komunikaci modul requests[14].

6.1.5 Lokální brána k systému firmy Logimic

Pro implementaci brány k rozhraní je využit jazyk TypeScript[15] a běhové prostředí Node.js[16], protože zbytek rozhraní a systému je implementován právě v tomto jazyce a běhovém prostředí.

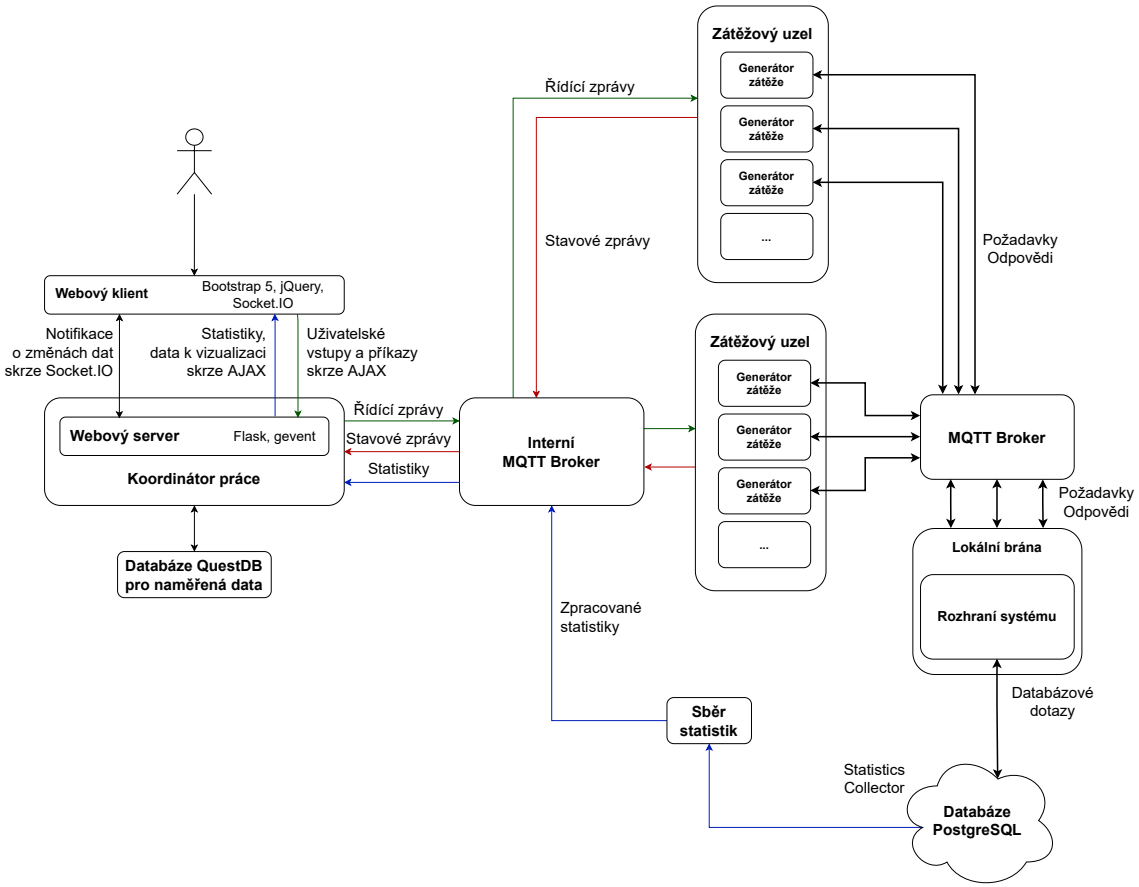
Aby bylo možné testovat souběžné přístupy do databázového systému, běží brána víceprocesově, k čemuž je využita vestavěná knihovna Cluster v prostředí Node.js[17]. Pro komunikaci pomocí protokolu MQTT je využita knihovna MQTT.js[18]. Fronta přijatých požadavků je pak implementována za pomoci knihovny Deque[19], ta nabízí oproti standardní implementaci pole v jazyce JavaScript operace nad frontou s konstantní časovou složitostí. Použil jsem ji, abych minimalizoval spotřebovaný výkon režií příjmu požadavků a nedocházelo ke zkreslením naměřených dat.

6.2 Detaily implementace komponent systému

Následující podsekce popisují implementační detaily jednotlivých komponent testovacího nástroje, které byly specifikovány v rámci návrhu v sekci 5.2. Konkrétně jde o koordinátor práce popsaný v podsekci 6.2.1 a s ním úzce spjaté webové rozhraní v podsekci 6.2.2. Dále pak podsekce 6.2.3 popisuje způsob implementace zátěžových uzlů, tento popis doplňuje podsekce 6.2.4 o generátorech zátěže. Podsekce 6.2.5 přibližuje způsob získávání statistik z databáze PostgreSQL. Na závěr je přiblížena struktura databáze pro ukládání nasbíraných dat v podsekci 6.2.6.

Obrázek 6.1 znázorňuje komponenty, jejich vzájemnou komunikaci a vazby a specifikuje u vybraných komponent použité technologie.

[14] https://docs.python-requests.org/en/latest/
[15] https://www.typescriptlang.org
[16] https://nodejs.org/en/
[17] https://nodejs.org/api/cluster.html
[18] https://github.com/mqttjs/MQTT.js
[19] https://github.com/petkaantonov/deque



Obrázek 6.1: Nákres implementačních komponent a jejich vazeb

6.2.1 Koordinátor práce

Jeho implementaci je obsažena v modulu `coordinator`. Je úzce integrován s webovým serverem, který poskytuje uživatelské rozhraní v podobě koncových bodů generujících komponenty webové stránky a přijímajících požadavky od uživatele.

Jeho hlavní komunikační součástí je instance MQTT klienta, který odebírá téma koordinátoru `/lgmcsim/coordinator`. Koordinátor pomocí MQTT klienta přijímá dva typy zpráv – stavové zprávy jednotlivých zátěžových uzlů a statistiky získané z databáze PostgreSQL. Koordinátor na základě zpráv aktualizuje aktuální stav uchovávaný v instanci třídy `SystemState`.

Třída SystemState

Datová struktura třídy `SystemState` je specifikována v podsekci 5.2.1, v následujících odstavcích se bude text věnovat implementačním detailům této třídy.

Seznam zátěžových uzlů v atributu `backends` je reprezentován instancemi třídy `BackendMeta`. Jde o informace o ID uzlu, seznam názvů podporovaných úloh, seznam generátorů a časovou značku. Pro ID uzlu se využívá UUID, aby byla zajištěna unikátnost. Seznam generátorů je reprezentován objekty třídy `WorkerMeta` (viz podsekci 6.2.4). Pole pro časovou značku slouží k uchování posledního času, kdy uzel zaslal stavovou informaci. Uzly, které zaslaly zprávu před delší dobou, než je specifikována v konfiguračním souboru, jsou skryty

a jsou považovány za nedostupné. Koordinátor si po dobu svého běhu uchovává informace o všech uzlech, které mu zaslaly nějaká data, aby mohl informace o nich opět obnovit a zobrazit uživateli, pokud by došlo pouze k výpadku uzlu a ne jeho ukončení.

Třída `SystemState` je zároveň zodpovědná za ukládání dat do databáze. V rámci její instance je spuštěno vlákno automatického ukládání, které každé 2 vteřiny vzorkuje stavová data a ukládá je do databáze. Dále poskytuje rozhraní pro manuální práci s databází (uložení, reset a inicializace databáze) a rozhraní pro zpracování dat, která jsou koordinátorovi zasílána z ostatních komponent.

Dále třída `SystemState` udržuje data nutná pro vykonávání testovacích scénářů, především jméno konfiguračního souboru s předpisem scénáře, mapování ID uzlů na pořadová čísla uzlů scénáře a globální pořadová čísla jednotlivých generátorů zátěže (více viz podsekci 6.2.4).

Manuální řízení

Uživatel je skrze rozhraní koordinátoru schopen řídit každý generátor v každém uzlu zvlášť i více generátorů ve více uzlech souběžně. Koordinátor zpracuje požadavek z webového rozhraní a rozešle do témat na základě ID uzlů řídící zprávy pro vykonání operací. Řídící zprávy mají formát JSON a obsahují objekt s atributy `cmd` s názvem příkazu pro uzel a `params`, který obsahuje objekt s pojmenovanými parametry příkazu.
Podporované řídící příkazy jsou následující:

- `spawn` – spustí nový generátor v uzlu,
- `kill` – ukončí určité generátory v uzlu podle jejich ID,
- `clear` – ukončí všechny generátory v uzlu,
- `taskset` – přiřadí specifikovaným generátorům úlohu s nastavenými parametry,
- `taskkill` – ukončí vykonávání úlohy na určených generátorech,
- `setint` – změní čas čekání mezi požadavky a tím změní intenzitu zasílání požadavků.

Automatické řízení podle scénáře

Testování je možné řídit podle scénáře, který je specifikován souborem ve formátu YAML (formát byl oproti jinde v aplikaci používanému formátu JSON zvolen pro tento účel, díky podpoře komentářů a obecně lepší lidské čitelnosti). Soubory popisující scénáře jsou načítány ze složky `scenarios` uvnitř modulu `coordinator`.

Koordinátor práce soubory načte, ověří, zda je popis scénáře kompletní a neschází žádná podstatná nastavení. Pokud je soubor v pořádku, tak pomocí postupného volání základních funkcí, které se využívají pro manuální řízení uvede uzly a generátory do požadovaného stavu a spustí zatěžování.

Konfigurační soubor se skládá ze dvou částí, resp. klíčů na nejvyšší úrovni. Povinná část `backends` je seznamem a specifikuje jednotlivé uzly. Koordinátor musí mít k dispozici pro spuštění scénáře dostatek uzlů, jinak se scénář nespustí. Prvky seznamu uzlů obsahují atributy `worker_groups`, ty specifikují skupiny generátorů, každá skupina pak musí obsahovat následující 3 atributy:

- `amount`, udávající počet generátorů ve skupině,

- `taskgrp`, specifikující jméno úlohy pro tyto generátory,
- `delay`, který definuje rozestup mezi požadavky a tím intenzitu zátěže.

Druhá část `options` může obsahovat atribut `acceleration`, kde je pomocí několika atributů specifikováno postupné zvyšování intenzity zátěže, pokud je požadováno. Postupné zrychlování slouží pro hledání množství požadavků, při kterém testované rozhraní přestane stíhat vyřizovat požadavky. V případě, že se v průběhu testu má intenzita postupně měnit, pak jsou rozestupy v atributech `delay` jednotlivých skupin generátorů brány jako základní a jsou pak ovlivněny nastavením zrychlování.
Atributy ovlivňující zrychlování jsou následující:

- `base_multiplier`, udávající násobek rozestupu na začátku testu,
- `goal_multiplier`, udávající násobek rozestupu na konci testu (musí být menší než na začátku, např. 1x na začátku a 0.1x na konci otestuje intenzitu zasílání požadavků od základní po desetinásobně rychlejší),
- `interval` udávající čas mezi změnami rychlosti,
- `steps_count` který udává v kolika krocích se rychlost změní od základní k cílové,
- `exclude`, nepovinný seznam, specifikuje jména úloh, kterých se zrychlování netýká, konkrétně například pravidelný výpočet KPIs

Výsledný konfigurační soubor vč. vysvětlujících komentářů může vypadat například jako na příkladu 6.1.

```
options:
  acceleration:
    # základní násobitel prodlev mezi požadavky - test začne 3x pomaleji
    base_multiplier: 3
    # finální násobitel prodlev - výsledná rychlost bude desetinásobná
    goal_multiplier: 0.1
    # postupně se bude zvyšovat intenzita každých 10 sekund
    interval: 10
    # intenzita se od nejpomalejší po nejrychlejší zvýší ve 120 krocích
    # zrychlovat se tedy bude 1200s (20 min)
    steps_count: 120
    # výpočet KPI má být pravidelný, tedy vyřazen ze změn intenzity
    exclude:
      - KPISim
backends:
  # na tomto uzlu poběží 200 generátorů simulujících IoT zařízení
  # zápis nových dat každých 7.5 sekundy (bez započtení zrychlování)
  - worker_groups:
      - amount: 200
        taskgrp: DeviceSim
        delay: 7.5
  # na tomto uzlu bude mít 1 generátor na starosti výpočet KPIs
  # dalších 99 generátorů pak simulaci IoT zařízení
  - worker_groups:
      - amount: 1
        taskgrp: KPISim
        delay: 300
      - amount: 99
        taskgrp: DeviceSim
        delay: 7.5
  - worker_groups:
      - amount: 50
        taskgrp: FNDSim
        delay: 20
```

Příklad 6.1: Konfigurační soubor scénáře `small-test`

Rozhraní s databází

Ačkoliv databázový systém QuestDB nabízí možnost komunikovat skrze rozhraní kompatibilní s knihovnami pro databázový systém PostgreSQL, jevilo se jako jednodušší využít nabízené REST API. Dotazy na databázi jsou tedy zasílány protokolem HTTP a databáze vrací výsledky ve formátu JSON, který je převeden na datový typ `dictionary` a výsledek je dále zpracován komponentou, která zaslala dotaz.

6.2.2 Webové rozhraní

Webové rozhraní tvoří na frontendu hlavní šablona definující rozložení a obsah jednotlivých panelů. Dynamicky měnící se komponenty stránky se načítají asynchronním voláním na webový server. Návrh a rozložení uživatelského rozhraní je přiblíženo v popisu a nákresech v podsekci 5.2.2. Zmíněné nákresy a popisy odpovídají výslednému implementovanému rozhraní.

Aktualizace komponent

Pravidelně jsou na stránce aktualizovány pouze grafy a statistiky. Grafy každých 5 vteřin, statistiky každé 2 vteřiny. Aktualizace ostatních součástí stránky (seznam generátorů, seznam a počet uzlů, stav ukládání dat do databáze) jsou vyvolány zasláním zprávy z koordinátora práce pomocí knihovny Socket.IO. Tento způsob obnovování těchto komponent jsem zvolil primárně, aby pravidelné aktualizace neobtěžovaly uživatele pravidelným problikáváním aktualizovaných elementů.

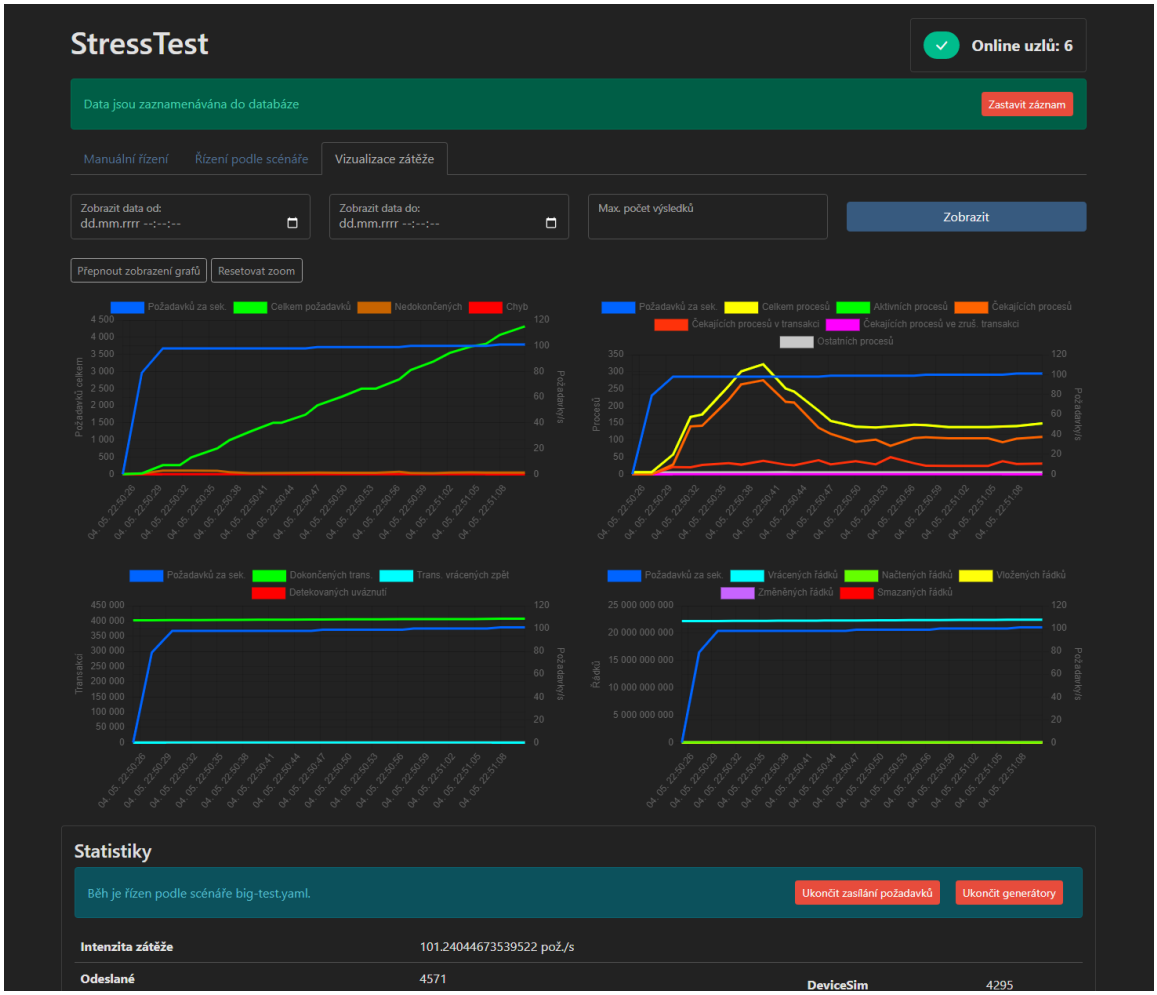
Pro účely dynamické aktualizace komponent rozhraní má webový server řadu koncových bodů, které na základě aktuálního stavu a vstupních parametrů generují HTML kód jednotlivých komponent. Pro volání koncového bodu a nahrazení HTML kódu v místě aktualizované komponenty je využívána knihovna jQuery.

Ovládání koordinátora práce

Při manuálním zasílání řídících zpráv o jednotlivých generátorech je pomocí javascriptové funkce `getSelected` na základě zvolených možností předzpracován seznam uzlů a generátorů, kterých se zpráva týká. Na základě tohoto seznamu pak koordinátor rozesílá řídící zprávy konkrétním uzlům.

Vykreslování grafů

Jak zmiňuje sekce 6.1.2, k vykreslování grafů je využita knihovna Chart.js pro jazyk JavaScript, která vykresluje grafy v prohlížeči uživatele. Data pro vykreslování získává z databáze a předzpracovává do formátu požadovaného knihovnou koordinátor práce. Ve výchozím stavu, při testování, se jedná o pravidelně aktualizovaná data za posledních 20 minut, jinak jde o data z časového úseku, který si zvolil uživatel. Na obrázku 6.2 lze vidět prostředí s vizualizací běžícího testování. Další screenshoty výsledného uživatelského rozhraní obsahuje příloha B.



Obrázek 6.2: Screenshot vizualizační části uživatelského rozhraní

6.2.3 Zátěžové uzly

Implementace zátěžových uzlů se nachází v modulu `backend`. Každý uzel si pro odlišení od ostatních při spuštění generuje unikátní ID ve formátu UUID4, připojí se ke specifikovanému MQTT brokeru a čeká na řídící zprávy (pro jejich výčet viz podsekci 6.2.1) od koordinátora práce.

Každý uzel aktivně zasílá koordinátoru práce zprávy o svém stavu, ty zahrnují jejich ID, seznam podporovaných úloh a seznam generátorů zátěže společně s daty o nich. Data o generátorech jsou reprezentována instancemi třídy `WorkerMeta`, kterou blíže specifikuje následující podsekce 6.2.4.

Mimo příjem zpráv uzly koordinují spouštění, ukončování a řízení jednotlivých generátorů zátěže skrze jejich rozhraní, blíže popsané taktéž v následující podsekci 6.2.4. Spouštění a ukončování uzlů odpovídá vytvářením a mazáním instancí objektů třídy `Worker`. Při spouštění uzel řídí přidělování názvů databázových spojení, které vyžaduje testované rozhraní. Pomocí specifikace názvů a počtu těchto názvů lze řídit počet databázových spojení, které testované rozhraní vytvoří. Tato vlastnost ale není uživatelsky konfigurovatelná, protože bylo pokusy (viz kapitolu 7, především podsekci 7.2.7) zjištěno, že testované rozhraní reaguje na počty těchto názvů neodpovídajícím zvýšením počtu reálných připojení

k databázovému server. Nesprávná konfigurace tedy může vést k odmítání připojení a tím k nesprávné funkčnosti testovacího nástroje a znehodnocení výsledků.

6.2.4 Generátory zátěže

Generátory zátěže jsou implementovány společně se zátěžovými uzly v modulu `backend`. Tvoří je instance třídy `Worker`. Ta obsahuje instanci klienta protokolu MQTT, který je využíván pro zasílání požadavků na testované rozhraní. Dále obsahuje plánovač úloh, jako jeho implementaci jsem zvolil modul APScheduler[20]. Plánovač úloh se využívá pro řízení rychlosti zasílání požadavků. Implementace řízení rychlosti zasílání požadavků pomocí plánovače úloh byla zvolena především díky snazší manipulaci a jednodušší implementaci oproti například implementaci za pomocí vláken a jejich uspávání. Rozhraní plánovače umožňuje úlohy snadno plánovat, přeplánovávat a automaticky udržuje přesné intervaly vykonávání.

Stav generátoru

Další stavové informace o generátoru jsou vyčleněny do instance třídy `WorkerMeta`, jejíž serializovaná verze je uzlem zasílána koordinátorovi práce. Dále třída zahrnuje rozhraní pro její řízení – spouštění, zastavování, změnu úlohy, změnu rychlosti zasílání požadavků.

Třída `WorkerMeta` obsahuje sadu informací o generátoru – jeho ID, ID uzlu, jméno vykonávané úlohy, jeho stav, rychlost zasílání požadavků a počty zaslaných a nevyřízených požadavků. Umožňuje tato data serializovat a zpětně deserializovat.

Úlohy pro zátěžové testy jsou reprezentovány samostatnými třídami, které specifikují konkrétně vykonávaný kód generující zátěž. Všechny tyto třídy musí být odvozené z bázové třídy `Task`. Tato bázová třída definuje základní princip generování zátěže a poskytuje rozhraní pro `Worker` a řídící aplikaci pro ovládání a monitorování průběhu vykonávání testovací úlohy. Na základě třídy `Task` je v případě potřeby možné implementovat další testovací úlohy, testující další části systému.

Požadavky a způsob jejich zasílání

Spuštění úlohy probíhá vytvoření instance třídy dané úlohy a případné konfigurace dalších parametrů, například postupného zrychlování a následně je úlohs spuštěna.

Časování zasílání požadavků zajišťuje zmíněný plánovač úloh. Po spuštění úlohy probíhá plánování ve dvou fázích, nejprve se za pomocí metody `first_run`, zajistí náhodné rozložení souběžně spouštěných úloh v čase. Poté se naplánuje periodické spouštění metody `run_once` v přednastaveném intervalu. Tato metoda obsahuje implementaci zaslání jednoho požadavku na testované rozhraní společně s poznamenáním čísla požadavku, za účelem sledování, zda byl vyřízen. Mimo to třída `Task` obsahuje metody pro podporu volitelného postupného zvyšování intenzity zátěže pomocí zkracování času mezi jednotlivými požadavky.

Operace, která bude v testovaném systému pomocí požadavku spuštěna, je vybírána z předdefinovaného seznamu předloh. Jedná se o instance třídy `Operation`, specifikují název operace v testovaném rozhraní a seznam parametrů, které testované rozhraní pro spuštění operace požaduje. Každý parametr pak má přiřazenu funkci, která generuje náhodný obsah parametrů požadavku.

[20] https://apscheduler.readthedocs.io/en/3.x/

Pro složitější operace lze z třídy `Operation` odvodit potomka, který může komplexnějším způsobem generovat a sestavit parametry požadavku. Tento postup je využit u generování požadavků IoT zařízení v třídě `UpdateDeviceOperation`.

Generátory si udržují přehled o počtu vyřízených požadavků na základě pořadových čísel, která jsou jednotlivým požadavkům přidělována a spolu s informací o uzlu a pořadovém čísle generátoru zasílána spolu s požadavkem bráně testovaného rozhraní. Brána předá zprávu rozhraní a po vyřízení požadavku zasílá konkrétnímu generátoru zprávu o dokončení požadavku, ten ji pak vyřadí z množiny nevyřízených požadavků.

Konkrétní implementace úloh pro testování

Konkrétní implementace úloh pro testování rozhraní obsahují třídy `DeviceSim`, `FNDSim` a `KPISim`.

Třída `DeviceSim` implementuje zátěž pocházející z IoT zařízení. Na základě globálního ID generátoru přistupuje generátor postupně ke 4 zařízením různých typů, pro které generuje náhodné hodnoty. Po vygenerování hodnot je složena zpráva s výběrem požadované operace v rozhraní (zápis do zařízení), jsou doplněny generované hodnoty a další nezbytné detaily, které vyžaduje rozhraní a zpráva se odešle.

Třída `FNDSim` využívá k vytěžování dvě operace testovaného rozhraní. První z nich získává z databáze informace o určitém KPI, protože tyto informace jsou zobrazovány v UI testovaného systému. Druhá z nich získává seznam detailů skupin zařízení, ta slouží k simulaci náročnější frontendové operace nad zařízeními.

Třída `KPISim` pouze pravidelně spouští operaci, která iniciuje přepočet všech KPI pro všechny zařízení. Tuto činnost v reálném systému provádí externí plánovač úloh, ten ale v lokálním prostředí není k dispozici. Navíc je tato operace tak náročná, že je výhodné mít při testování přehled o tom, kdy byla tato operace spuštěna, aby bylo možné pozorovat její vliv na systém.

6.2.5 Sběr statistik z databáze PostgreSQL

Skript pro sběr statistik z databáze v cyklu čte hodnoty z tabulek `pg_stat_activity` a `pg_stat_database` v databázi testovaného systému. Jedná se o tabulky kolektoru statistik, do kterých automaticky databázový systém PostgreSQL ukládá počty spuštěných databázových procesů spolu s jejich stavem (zda vykonávají transakci, čekají, čekají při vykonávání transakce, apod.), dále počty dokončených a zrušených transakcí, detekovaných deadlocků a počty řádků, které byly v databázi modifikovány, vloženy, smazány, z databáze vybrány a kolik jich bylo po dotazu vráceno klientům.

V současné době využívá systém firmy Logimic starší verzi PostgreSQL, která mezi statistikami neobsahuje informace o délce vykonávání dotazů, proto tyto statistiky nejsou nikde v této práci zahrnuty.

Hodnoty získané z těchto databázových tabulek skript zpracuje do JSON formátu a skrze MQTT broker zašle koordinátorovi práce.

6.2.6 Databáze pro ukládání nasbíraných dat

Skládá se ze dvou tabulek: `stats` a `task_stats`. Tabulka `stats` obsahuje data o průběhu testování, která jsou platná k jisté časové značce. Záznamy v tabulce `task_stats` tento obecný pohled doplňují informacemi, které lze shromáždit o jednotlivých typech zátěžových úkolů odděleně.

Součástí vzorku v tabulce `stats` je počet celkem zaslaných požadavků, prozatím nedokončených požadavků, požadavků s chybovou odpovědí, rychlost zasílání požadavků, dále pak data o databázových procesech – počet procesů, kolik z toho je zaneprázdněných a případně čím (vykonávající transakci, čekající v transakci, čekající ve zrušené transakci a ostatní kategorie čekání), dále počty dokončených a vrácených transakcí a detekovaných uváznutí a na závěr pak počty řádků databáze, které byly vybrány, vráceny klientům, vloženy, změněny a smazány. Tabulka `task_stats` obsahuje pro tyto body v čase záznamy k jednotlivým spuštěným úlohám, konkrétně jejich podíl na rychlosti zasílání požadavků a na množství zaslaných a nedokončených požadavků.

Kapitola 7

Testování

Tato kapitola popisuje v sekci 7.1 způsoby, jakým byla ověřena správná funkčnost aplikace a naplnění požadavků a dále v sekci 7.2 přibližuje výsledky použití aplikace v praxi při testování nasazení rozhraní na několik různě výkonných strojů.

7.1 Ověření funkčnosti

Jak již bylo zmíněno v kapitole 6 o implementaci, pro komunikaci komponent aplikace i generátorů zátěže s bránou rozhraní se využívá protokol MQTT. Pro ověření funkčnosti a naplnění hlavních požadavků na nástroj bylo možné využít aplikaci MQTT Explorer[1].

Požadavek na produkování zátěže odpovídající specifikovaným hlavním zdrojům bylo ověřeno zachytáváním a kontrolou obsahu MQTT zpráv mezi koordinátorem a zátěžovými uzly a MQTT zpráv, kterými brána testovaného rozhraní zasílá generátorům zátěže zpět odpovědi na zaslané požadavky. Konfigurovatelnost zátěže byla ověřena pozorováním rozestupů mezi zasílanými zprávami a srovnáním s nakonfigurovanými hodnotami. Rozdělitelnost zdrojů zátěže bylo ověřeno spuštěním uzlů v několika virtuálních strojích a připojením k jednomu message brokeru. Následně byly obdobným způsobem jako u předchozího testování odchyceny zprávy a zkontrolován jejich původ a obsah. Shromažďování dat bylo testováno manuálními kontrolami databáze vůči datům reprezentovaným v rozhraní systému.

7.2 Využití nástroje k testování rozhraní

Podsekce 7.2.1 popisuje způsob testování a přibližuje hardwarové a softwarové vybavení použité pro testy. Podsekce 7.2.3 a 7.2.4 popisují konkrétní konfigurace, testovací scénář a specifika testů v lokálním prostředí. Testy na vzdálených databázích přibližují spolu s konfiguracemi a dalšími detaily podsekce 7.2.5 a 7.2.6. Podsekce 7.2.7 pak obsahuje tabulku s výsledky testování a rozebírá další zjištěné poznatky.

Každá podsekce, popisující konkrétní test obsahuje screenshot s vizualizací průběhu tohoto testu. V grafu požadavků je označen bod, od kterého se fronta požadavků pouze zvětšovala, tedy bod maximální zátěže, kterou by byl systém schopen zpracovat. Screenshoty jsou barevně invertovány a upraveny pro tisk.

[1] http://mqtt-explorer.com

7.2.1 Testované konfigurace a podmínky testování

Lokální fáze testování využívala vždy dva souběžně běžící virtuální počítače vytvořené a provozované pomocí aplikace Oracle VM VirtualBox[2] ve verzi 6.1.26. Hostitelský počítač byl vybaven procesorem AMD Ryzen 5 5600X, 32GB operační paměti o frekvenci 3000MHz a OS Windows 10 21H2. Všechny počítače jako operační systém využívaly OS Debian Linux[3] ve verzi 11 bez grafického prostředí.

Všechny běhy v lokální fázi testování shodně využívaly pro běh brány a rozhraní firmy Logimic konfiguraci s 4 logickými procesory a 4096 MB operační paměti, tato konfigurace se ukázala být v lokálních podmínkách dostatečná pro všechny testy.

Fáze testování na vzdálených databázích sestávala ze dvou testů. První z nich využívá shodnou konfiguraci databáze, jako využívá aktuální demonstrační nasazení systému firmy Logimic. Druhý z nich proběhl na databázovém serveru, který byl nasazen v instanci virtuálního serveru Oracle Compute Ampere A1[4].

U všech testů byl databázový server PostgreSQL zkonfigurován, aby velikost cache (parametr `shared_buffers`) odpovídala doporučené[5] velikosti, tedy 25% operační paměti.

7.2.2 Použité testovací scénáře

K testování byly použity dva scénáře, které přibližně vychází z konzultovaných množství uživatelů a zařízení, jejich poměrů a odhadů jejich intenzity.

První z nich je specifikován v souboru `big-test.yaml`. Scénář vyžaduje 6 zátěžových uzlů, na kterých spustí 800 generátorů zátěže. Z tohoto počtu je jeden vyhrazen pro periodické spouštění výpočtu KPIs každých 300 sekund. 699 je jich vyhrazeno pro simulaci zátěže IoT zařízení zasíláním jednoho požadavku každých 7.5 sekundy a 100 pro simulaci frontendové zátěže zasíláním požadavků každých 20 sekund.

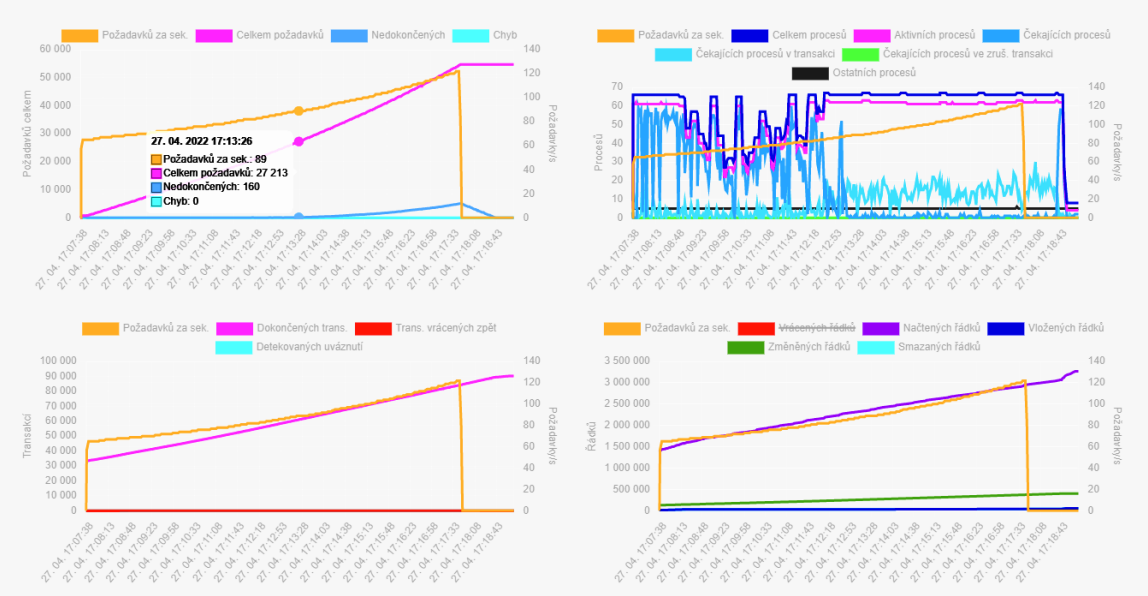
Druhý ze scénářů vyžaduje pouze 3 zátěžové uzly a spouští 350 generátorů zátěže. 299 z nich generuje 1x za 7.5 sekundy zátěž simulující IoT zařízení, 50 z nich generuje 1x za 20 sekund zátěž simulující frontendové přístupy a 1 je opět vyhrazen pro spouštění výpočtu KPIs 1x za 300 sekund.

7.2.3 Test č. 1 – lokální

Test probíhal lokálně na virtuálním počítači. Virtuální server měl k dispozici 1 logický procesor, 1280 MB RAM. Tato konfigurace by měla být odpovídající používané konfiguraci v demonstračním nasazení (konkrétně se jedná o Amazon AWS RDS instanci typu db.t2.micro, viz podsekce 7.2.5). K testování byl využit scénář `big-test.yaml`.

Na obrázku 7.1 lze pozorovat průběh testu. Na pravém horním grafu procesů je patrná reakce databázového systému. Tam se po dosažení maximální zátěže (vyznačený bod vlevo nahoře) ustálil kolísající počet procesů databázového serveru na dostupném maximu, při kterém bylo dostupné procesorové jádro plně vytíženo. Po ukončení generování zátěže je fronta rychle vyřízena a procesy jsou ukončeny.

[2] https://www.virtualbox.org
[3] https://www.debian.org
[4] https://www.oracle.com/cloud/compute/arm/
[5] https://www.postgresql.org/docs/current/runtime-config-resource.html

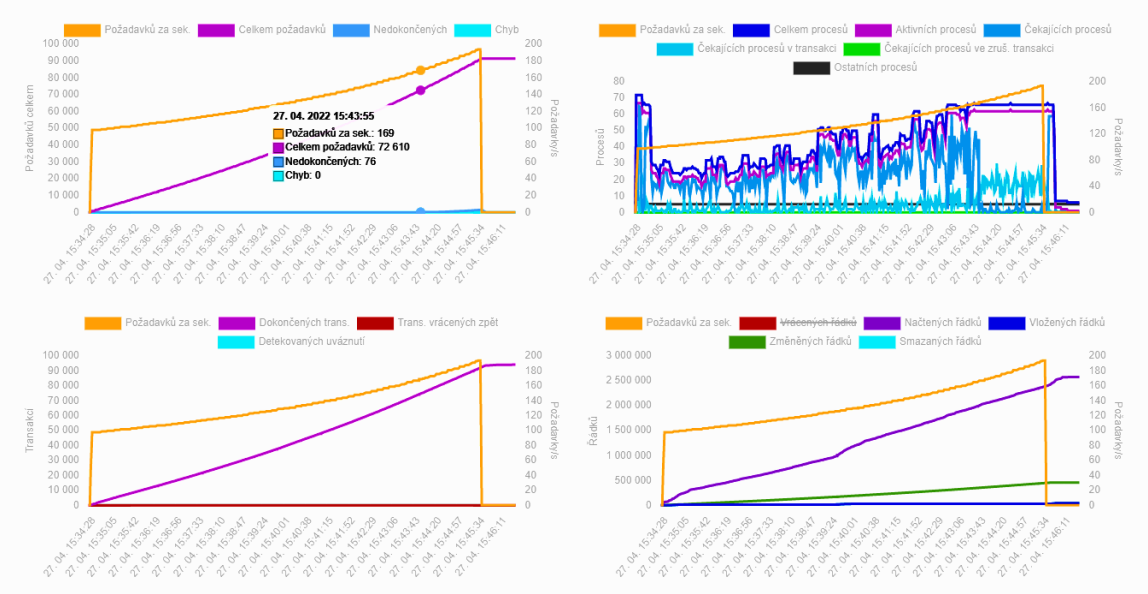


Obrázek 7.1: Průběh testu č. 1

7.2.4 Test č. 2 – lokální

Test probíhal lokálně na virtuálním počítači. Virtuální server měl k dispozici 2 logické procesory, 1280 MB RAM. Tato konfigurace by měla odpovídat Amazon AWS RDS instanci typu db.t3.micro. K testování byl využit scénář `big-test.yaml`.

Na obrázku 7.2 lze pozorovat pozvolnější růst zátěže systému než u testu č. 1, test navíc začínal s vyšší počáteční intenzitou. V pravém horním grafu procesů lze pozorovat pozvolný nárůst počtu běžících procesů společně s narůstající zátěží. Počet procesů se opět po dosažení maximální zátěže ustálil na obdobném počtu jako u testu č. 1 a zvýšil se počet procesů čekajících v průběhu transakcí. Zhruba po 5 minutách testu je patrný citelný nárůst v počtu řádků, které databázový server předal klientům. Podle záznamů tomuto nárůstu odpovídá spuštění výpočtu KPI.



Obrázek 7.2: Průběh testu č. 2

7.2.5 Test č. 3 – AWS RDS

Test probíhal na vzdálené databázi Amazon AWS RDS v datacentru s umístěním v USA. Databáze má typ instance db.t2.micro[6], což v době testování odpovídalo 1 jádru v rámci nespecifikovaného procesoru Intel a 1 GB paměti.

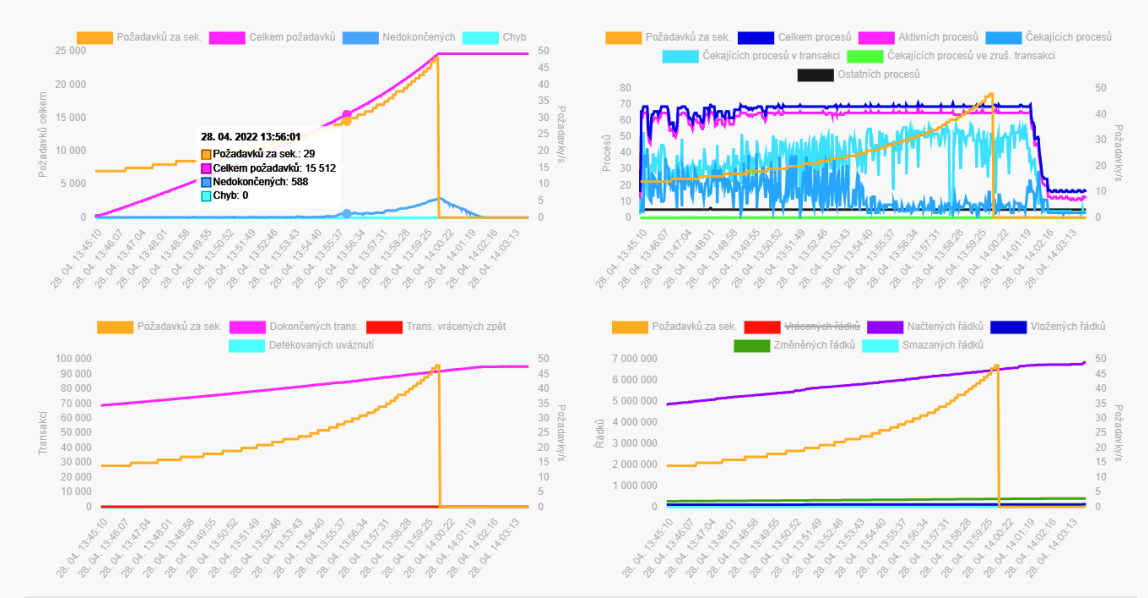
K testování byl nejprve využit scénář `big-test.yaml` podobně jako u předchozích dvou testů, ale rozhraní nestíhalo vyřizovat požadavky už při spuštění testu. Proto byl využit odlišný scénář `small-test.yaml`, který využívá méně uzlů, generátorů a byl mu nakonfigurován postupnější náběh.

V průběhu testování se, na rozdíl od lokálních testů, udržovala konstantně fronta nevyřízených požadavků, kterou přisuzuji zpoždění sítě. Na výsledcích testování (viz tabulku 7.2) lze pozorovat, že reálná databáze v cloudovém prostředí je výrazně pomalejší oproti testu č. 1 (viz podsekci 7.2.3). Tento fakt nasvědčuje, že hardware použitý pro běh databáze v cloudovém prostředí AWS je méně výkonný a označování výkonnostních stupňů počtem jader nespecifikovaného procesoru je zavádějící.

Během testování bylo dále zjištěno, že při použití vzdálené databáze se výpočet KPI pro několik tisíc uložených zařízení nestíhá dokončit před začátkem dalšího plánovaného přepočtu. Na základě konzultací se jako pravděpodobná příčina jeví způsob implementace, při kterém je velmi velké množství SQL dotazů postupně zasíláno na vzdálený server. Výsledky taktéž naznačují, že tato konfigurace databáze nedostačuje pro plánované rozšíření demonstračního nasazení, ve kterém by měly být v provozu řádově tisíce zařízení.

Obrázek 7.3 zobrazuje průběh třetího testu. Oproti předcházejícím testům je patrný vysoký počet procesů už od začátku testování. V maximu se počet stabilizuje ještě před přetížením systému, které podle levého horního grafu přicházelo pozvolněji než u předchozích testů.

[6] https://aws.amazon.com/rds/instance-types/



Obrázek 7.3: Průběh testu č. 3

7.2.6 Test č. 4 – Virtuální server

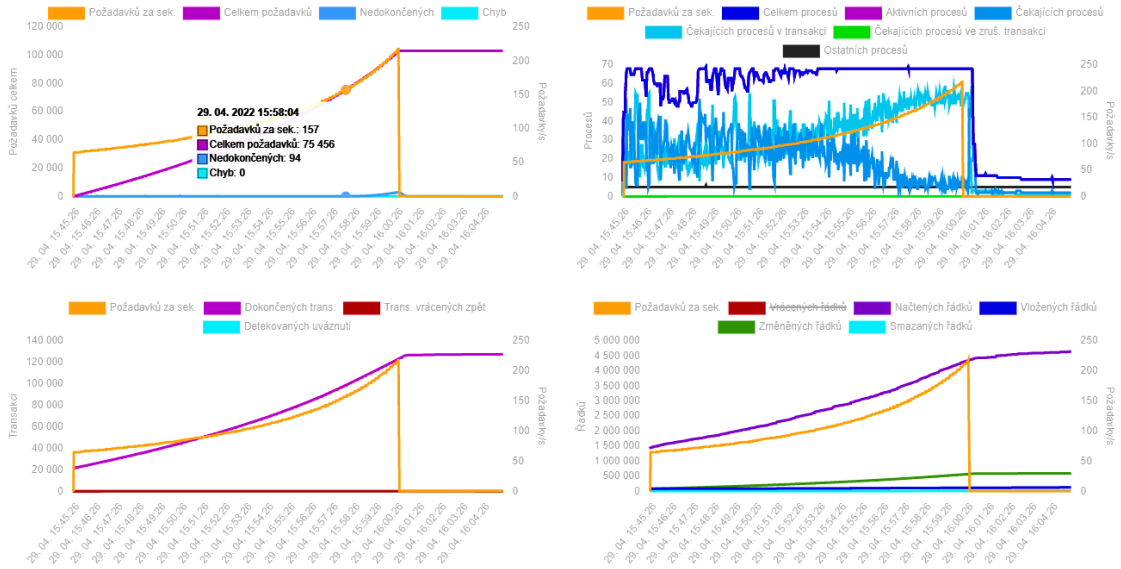
Test probíhal na vzdáleném virtuálním serveru Oracle Compute Ampere A1 s umístěním v datacentru v Německu. Virtuální server měl k dispozici 4 logické procesory Ampere Altra a 24 GB RAM. K testování byl, obdobně jako u lokálních testů využit scénář `big-test.yaml`.

Tento testovací HW jsem se rozhodl využít k provedení jednoho z testů jako srovnání s AWS RDS. Oproti RDS, které poskytuje databázi jako službu, se jedná o manuální nasazení databázového systému na vzdálený virtuální server. Přes zásadní rozdíly ve výkonu (viz tabulku 7.2) obě tyto služby patří u svých poskytovatelů k úrovním poskytovaným zdarma.

Velký rozdíl je mezi použitými servery Amazonu a Oraclu v geografické poloze a tím i v odezvě od rozhraní. Tento rozdíl se zásadním způsobem ve výsledcích testování neprojevil. Taktéž se udržovala fronta nevyřízených požadavků, řádově podobné velikosti.

Při testování nastal podobný jev jako u testu č. 3 (viz podsekci 7.2.6). Přestože server leží v Evropě a má výrazně vyšší výkon, se výpočty KPIs nestíhají dokončovat dříve, než začnou výpočty nové. Výsledky ale naznačují, že server využívající tento hardware by měl dostačovat pro použití po rozšíření aktuálního demonstračního nasazení na několik tisíc zařízení.

Obrázek 7.4 zobrazuje průběh čtvrtého testu. Na grafech je patrné podobné chování, jako v případě předcházejících testů. Hlavní odlišností je dosažený maximální výkon.



Obrázek 7.4: Průběh testu č. 4

7.2.7 Shrnutí výsledků testování

Testovací nástroj zatěžuje rozhraní podle dané konfigurace a splňuje tím požadavky specifikované v podsekci 4.2. Za pomoci nástroje byly provedeny 4 demonstrační experimenty na různých hardwarových konfiguracích. Parametry experimentů jsou shrnuty v tabulce 7.1 a výsledky těchto experimentů jsou zaznamenány v tabulce 7.2. Na obrázku 7.5 jsou v grafu srovnány maximální dosažené úrovně zátěže.

Rozdíly v počátečních intenzitách zátěže ve výsledkové tabulce byly způsobeny změnami násobitele ve scénáři. Tyto změny se děly za účelem zkrácení testu tam, kde bylo možné předpokládat, že rozhraní odbaví více požadavků.

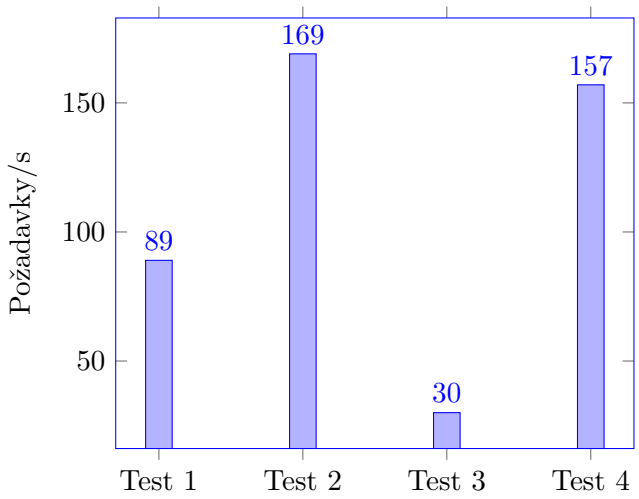
Při testování byly požadavky obvykle zasílány rychleji a z menšího množství generátorů zátěže, než by se dělo v reálném nasazení. Z konzultací vyplynulo, že IoT zařízení zasílají standardně svůj stav zhruba 1x za minutu, proto tabulka 7.2 uvádí i odhadovaný počet zařízení, které by bylo možné s touto konfigurací obsloužit.

Číslo testu	Umístění	Počet log. procesorů	Paměť	Scénář
1	lokální	1	1280 MB	`big-test`
2	lokální	2	1280 MB	`big-test`
3	AWS (USA)	1	1024 MB	`small-test`
4	Oracle (DE)	4	24 576 MB	`big-test`

Tabulka 7.1: Tabulka testů

Číslo testu	Výchozí zátěž [pož./s]	Max. zátěž [pož./s]	Zátěž ze zařízení [pož./s]	Odhad. počet zařízení
1	65	89	85	5 000
2	98	169	157	9 400
3	14	30	28.3	1 700
4	65	157	149	8 900

Tabulka 7.2: Výsledky testů



Obrázek 7.5: Maximální zátěž na rozhraní

Zjištěné závěry

Test č. 3 (podsekce 7.2.5) ukázal, že oproti předpokladům zjištěným testem č. 1 (podsekce 7.2.3) je výkon v rámci AWS RDS instance db.t2.micro nízký. Při srovnatelném počtu procesorových jader a paměti bylo rozhraní v lokálním testu schopno obsloužit 3x více požadavků než v testu se vzdálenou databází. Popis výkonu databáze podle těchto parametrů je dle zjištěných výsledků ze strany provozovatele cloudové platformy zavádějící. Toto tvrzení dále podpořilo srovnání testu č. 2 (podsekce 7.2.4) a testu č. 4 (podsekce 7.2.6), které dosáhly řádově podobných výsledků. Test č. 2 jich ale dosáhl za použití dvou logických jader procesoru AMD Ryzen, zatímco test č. 4 za použití čtyř jader procesoru Ampere A1.

Z těchto zjištění plyne, že predikovat a porovnat kapacitu systému napříč systémy je s omezenými informacemi o hardwaru, které poskytovatelé zveřejňují, velmi náročné. Jako

spolehlivější se jeví praktické testování konkrétního hardwaru u konkrétního poskytovatele a následná úprava konfigurace na základě výsledků.

Testování dále ukázalo, že případné rozšiřování demonstračního nasazení systému řádově na jednotky tisíc zařízení by znamenalo nutnost vylepšení databázového serveru, ať už změnou tarifu nebo nasazením u jiného poskytovatele. Jako vhodný kandidát pro testování při využití výkonnostní úrovně poskytované zdarma se jeví Oracle Compute Ampere A1 server, použitý v testu č. 4. Nevýhodou tohoto řešení je, že databáze zde nasazená je nasazená manuálně na cloudové službě typu IaaS oproti spravované automaticky nasazené a udržované databázové službě typu SaaS u AWS[7]. Výhodou je poměrně štědré množství systémových zdrojů poskytované k tomuto typu serveru a dostatek výkonu pro tisíce zařízení.

Všechny 4 testy skončily v této konfiguraci vyčerpáním dostupného výkonu procesoru, žádný z nich nevyčerpal dříve operační paměť. K vyčerpání operační paměti došlo pouze při experimentování s konfiguracemi, využívajících více databázových spojení, než se ve výchozím stavu využívá. V souvislosti s tímto chováním byl při experimentování během vývoje odhalen problém s množstvím databázových připojení testovaného rozhraní.

Rozhraní systému využívá vytvořená databázová spojení na základě jména, které je specifikováno klientem při zasílání požadavku. Při využití více různých jmen, které rozhraní nijak nelimituje, se zvyšuje výkon a s ním využití paměti databázovým serverem. Okamžitě ale dojde ve výchozím nastavení k vyčerpání dostupných databázových připojení. Dostupných připojení je ve výchozí konfiguraci PostgreSQL 100. Rozhraní jich však při jednotkách použitých jmen využívá stovky a při desítkách jmen tisíce. Při navýšení počtu připojení na stroji s nedostatkem operační paměti toto chování způsobilo násilné ukončení databázového serveru operačním systémem. Testy byly s ohledem na tuto skutečnost prováděny pouze s jedním jménem připojení a tedy limitovaným počtem připojení k databázi. Výsledky testů ukazují, že limit na jedno jméno pravděpodobně limitoval počet databázových procesů na podobné hranici, zatímco při experimentech s více jmény dosahovaly počty databázových procesů stovek. Oprava této chyby a využití více připojení by mohlo přispět k efektivnějšímu využití dostupných prostředků a dosažení vyšších výkonů.

[7] Typy cloudových služeb viz podsekce 2.2.1.

Kapitola 8

Závěr

Cílem této práce bylo vytvořit nástroj pro zátěžové testování systému pro správu chytrých měst firmy Logimic a především relační databáze, kterou tento systém používá pro uchovávání dat. Hlavním účelem nástroje je pak usnadnit odhad hardwarových nároků pro budoucí nasazení systému a pomoci ověřit kapacitu nasazeného systému před ostrým spuštěním.

Před tvorbou tohoto nástroje bylo nutné nastudovat problematiku internetu věcí, databázových systému a interní specifika testovaného systému. Následně byla na základě interní dokumentace a konzultací specifikována sada požadavků, které by nástroj měl splnit. S ohledem na tuto specifikaci byl následně proveden průzkum existujících řešení, na jehož základě bylo rozhodnuto vytvořit vlastní nástroj na míru zmíněného systému. Podle specifikací pak byl nástroj navržen a implementován.

Výsledkem práce je konfigurovatelný testovací nástroj s možností škálování pro rozdělení zátěže na více zdrojových míst a zvýšení maximální zátěže, kterou lze generovat. Nástroj umožňuje testování systému pro správu chytrých měst a umožňuje během testování monitorovat databázový systém PostgreSQL, kterou tento systém využívá. Testovací nástroj poskytuje webové uživatelské rozhraní. To umožňuje uživateli manuálně řídit generování zátěže, spouštět předdefinované scénáře generování zátěže a pozorovat, jak systém na zátěž reaguje. Data o zátěží umožňuje systém zobrazit i zpětně a po proběhlém testu analyzovat výsledky.

Nástroj bude využit ve firmě Logimic pro testování nasazení systému před ostrým nasazením do provozu. Pomůže tím firmě před spuštěním dalších fází demonstračního nebo i ostrého nasazení odhadnout, kolik výkonu bude systém potřebovat. Případně nástroj pomůže po nasazení vyzkoušet, zda jsou servery na očekávanou zátěž připraveny. Již nyní bylo pomocí nástroje zjištěno, že současná konfigurace databázového serveru pro příští fázi rozšiřování nedostačuje. Nástroj také bude možné využít pro testování budoucích optimalizací a aktualizací systému a jejich vlivu na zátěž databázového serveru.

Do budoucna by bylo možné aplikaci rozšířit o další způsoby a možnosti testování jednotlivých součástí např. automatizovaným procházením frontendu systému. Rozšířit možnosti sběru dat například o údaje o vytížení operačního systému počítače s databázovým serverem nebo rozhraním aplikace. Případně rozšířit uživatelské rozhraní například o možnosti interaktivní tvorby scénářů bez nutnosti tvořit konfigurační soubor nebo správu předešlých testů.

Literatura

[1] *MQTT Version 5.0*. Andrew Banks, Ed Briggs, Ken Borgendale, Rahul Gupta. OASIS Standard., březen 2020 [cit. 2021-12-18]. Dostupné z: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

[2] Amazon Web Services, Inc. *AWS IoT Core*. [cit. 2022-04-16]. Dostupné z: https://aws.amazon.com/iot-core/.

[3] Amazon Web Services, Inc. *Cloud Services - Amazon Web Services (AWS)*. [cit. 2022-01-22]. Dostupné z: https://aws.amazon.com/.

[4] Amazon Web Services, Inc. *Fully Managed Relational Database - Amazon RDS - Amazon Web Services*. [cit. 2022-03-12]. Dostupné z: https://aws.amazon.com/rds/.

[5] Amazon Web Services, Inc. *Purpose-Built Databases on AWS | Amazon Web Services*. [cit. 2022-03-12]. Dostupné z: https://aws.amazon.com/products/databases/.

[6] Amazon Web Services, Inc. *Serverless Computing - AWS Lambda - Amazon Web Services*. [cit. 2022-01-22]. Dostupné z: https://aws.amazon.com/lambda/.

[7] Amazon Web Services, Inc. *What Is Amazon DynamoDB? - Amazon DynamoDB*. [cit. 2022-03-12]. Dostupné z: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html.

[8] Antonopoulos, N. a Gillam, L. *Cloud computing*. Springer, 2010.

[9] Apache Software Foundation. *Apache Jmeter*. [cit. 2022-04-18]. Dostupné z: https://jmeter.apache.org.

[10] Artillery Software Inc. *Artillery Docs*. [cit. 2022-04-18]. Dostupné z: https://www.artillery.io/docs.

[11] Artillery Software Inc. *Pricing - Artillery.io*. [cit. 2022-04-18]. Dostupné z: https://www.artillery.io/pricing.

[12] Bayer, M. SQLAlchemy. In: Brown, A. a Wilson, G., ed. *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. Aosabook.org, 2012. Dostupné z: http://aosabook.org/en/sqlalchemy.html.

[13] Bor, M., Vidler, J. E. a Roedig, U. LoRa for the Internet of Things. Junction Publishing. 2016.

[14] Craig, I. *The interpretation of object-oriented programming languages*. Springer Science & Business Media, 2001.

[15] Google Ireland Limited. *Google Cloud Databases*. [cit. 2022-04-13]. Dostupné z: https://cloud.google.com/products/databases.

[16] Google Ireland Limited. *Google Fit*. [cit. 2022-01-19]. Dostupné z: https://www.google.com/fit/.

[17] Grafana Labs. *K6 Documentation*. [cit. 2022-04-18]. Dostupné z: https://k6.io/docs/.

[18] Grafana Labs. *Plan & Pricing | k6 Cloud*. [cit. 2022-04-18]. Dostupné z: https://k6.io/pricing/.

[19] Greengard, S. *The Internet of Things*. MIT Press, 2015.

[20] Hipp, R. D. et al. *SQLite*. [cit. 2022-03-12]. Dostupné z: https://www.sqlite.org/index.html.

[21] Hornby, A. S. *Cloud-computing noun*. Oxford University Press [cit. 2022-01-21]. Dostupné z: https://www.oxfordlearnersdictionaries.com/definition/english/cloud-computing.

[22] Hornby, A. S. *Internet of things*. Oxford University Press [cit. 2022-01-20]. Dostupné z: https://www.oxfordlearnersdictionaries.com/definition/english/internet-of-things.

[23] InfluxData, Inc. *Flux 0.x Documentation*. [cit. 2022-03-12]. Dostupné z: https://docs.influxdata.com/flux/v0.x/.

[24] InfluxData, Inc. *InfluxDB design principles | InfluxDB OSS 2.1 Documentation*. [cit. 2022-03-12]. Dostupné z: https://docs.influxdata.com/influxdb/v2.1/reference/key-concepts/design-principles/.

[25] InfluxData, Inc. *InfluxDB pricing - InfluxData*. [cit. 2022-04-12]. Dostupné z: https://www.influxdata.com/influxdb-pricing/.

[26] Ireland, C., Bowers, D., Newton, M. a Waugh, K. Understanding object-relational mapping: A framework based approach. *International lournal On Advances in Software*. Citeseer. 2009, sv. 2, č. 2.

[27] Jamsa, K. *Cloud computing: SaaS, PaaS, IaaS, virtualization, business models, mobile, security and more*. Jones & Bartlett Publishers, 2012.

[28] Levene, M. a Loizou, G. *A Guided Tour of Relational Databases and Beyond*. London: Springer London, 1999. ISBN 9781852330088 9780857293497. Dostupné z: http://link.springer.com/10.1007/978-0-85729-349-7.

[29] MariaDB. *What is MariaDB Galera Cluster*. [cit. 2022-04-13]. Dostupné z: https://mariadb.com/kb/en/what-is-mariadb-galera-cluster/.

[30] MariaDB Foundation. *MariaDB Foundation*. [cit. 2022-03-15]. Dostupné z: https://mariadb.org.

[31] Meier, A. a Kaufmann, M. *SQL & NoSQL databases*. Springer, 2019.

[32] Merriam-Webster Inc.. *Cloud computing*. Merriam-Webster [cit. 2022-01-21]. Dostupné z: https://www.merriam-webster.com/dictionary/cloud%20computing.

[33] Merriam-Webster Inc.. *Internet of Things*. Merriam-Webster [cit. 2022-01-18]. Dostupné z: https://www.merriam-webster.com/dictionary/Internet%20of%20Things.

[34] Microsoft Corporation. *Azure databases - Types of databases on Azure*. [cit. 2022-04-13]. Dostupné z: https://azure.microsoft.com/en-ca/product-categories/databases/.

[35] OASIS. *About us - OASIS*. [cit. 2021-12-10]. Dostupné z: https://www.oasis-open.org/org/.

[36] O'Neil, E. J. Object/Relational Mapping 2008: Hibernate and the Entity Data Model (Edm). In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2008, s. 1351–1356. SIGMOD '08. DOI: 10.1145/1376616.1376773. ISBN 9781605581026. Dostupné z: https://doi.org/10.1145/1376616.1376773.

[37] Oracle. *MySQL 8.0 Reference Manual*. [cit. 2022-03-15]. Dostupné z: https://dev.mysql.com/doc/refman/8.0/en/.

[38] Porter, M. E. a Heppelmann, J. E. How smart, connected products are transforming competition. *Harvard business review*. 2014, sv. 92, č. 11, s. 64–88.

[39] QuestDB. *QuestDB: the database for time series*. [cit. 2022-04-12]. Dostupné z: https://questdb.io/docs/introduction/.

[40] Schwab, K. *The Fourth Industrial Revolution*. Penguin Random House, 2017.

[41] The Postgres-XL Global Development Group. *Postgres-XL 10r1.1 Documentationn*. 2022.

[42] The Postgres-XL Global Development Group. *PostgreSQL 14.2 Documentation*. 2022.

[43] TypeORM. *TypeORM*. [cit. 2022-02-22]. Dostupné z: https://typeorm.io/.

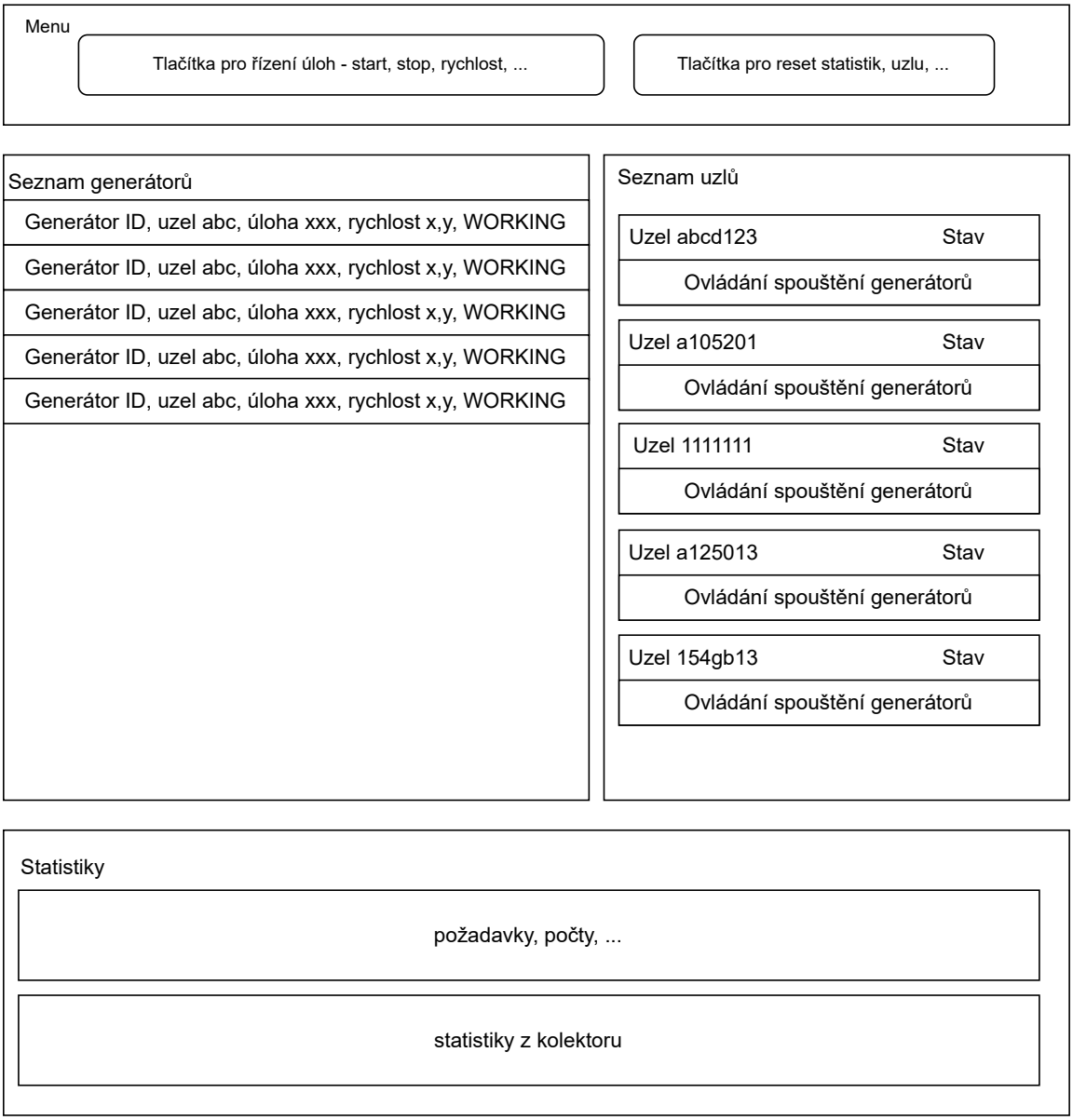
[44] Van Eyk, E., Toader, L., Talluri, S., Versluis, L., Uță, A. et al. Serverless is more: From paas to present cloud computing. *IEEE Internet Computing*. IEEE. 2018, sv. 22, č. 5, s. 8–17.

[45] Wortmann, F. a Flüchter, K. Internet of things. *Business & Information Systems Engineering*. Springer. 2015, sv. 57, č. 3, s. 221–224.

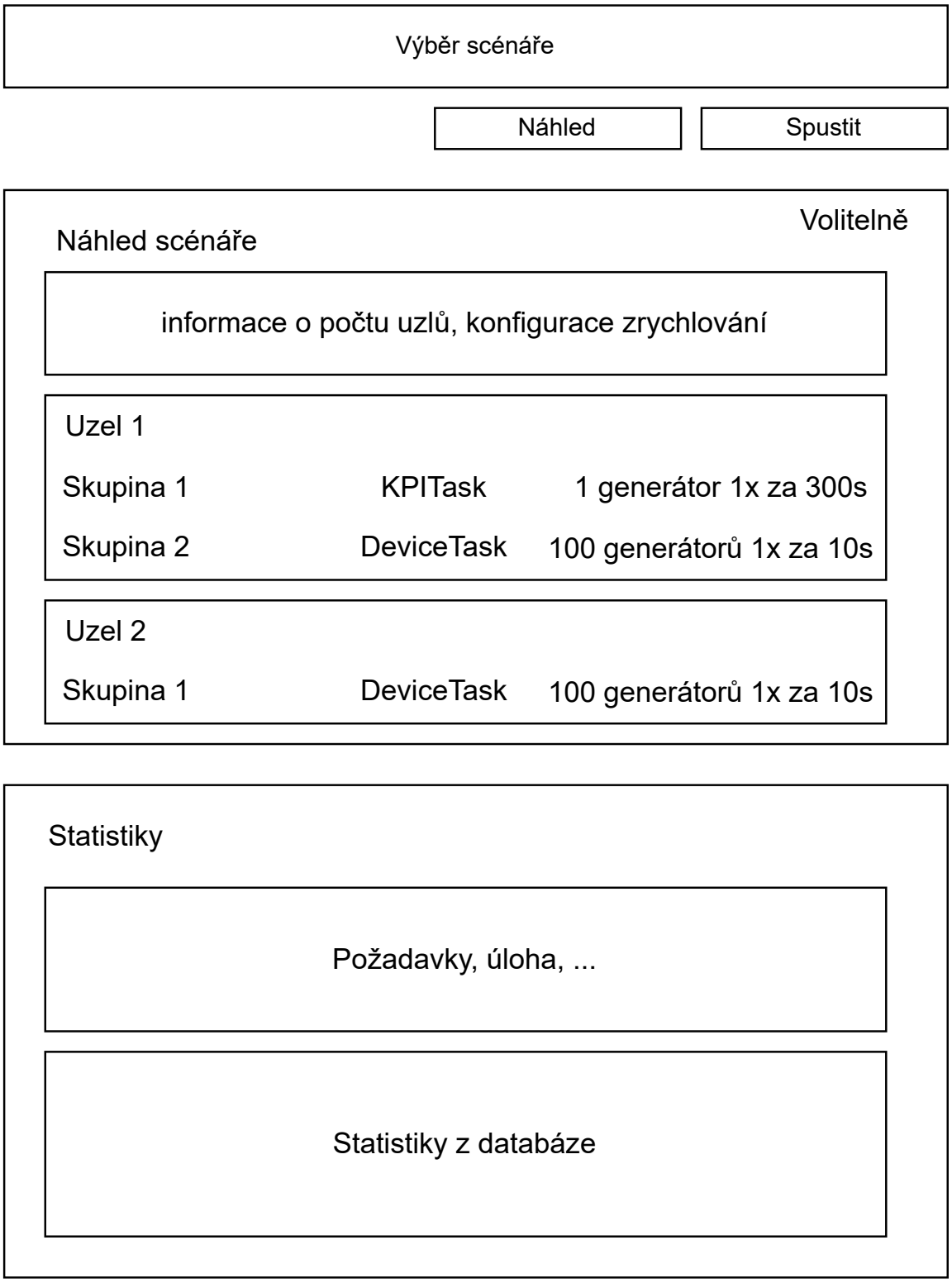
Příloha A

Návrhy uživatelského rozhraní

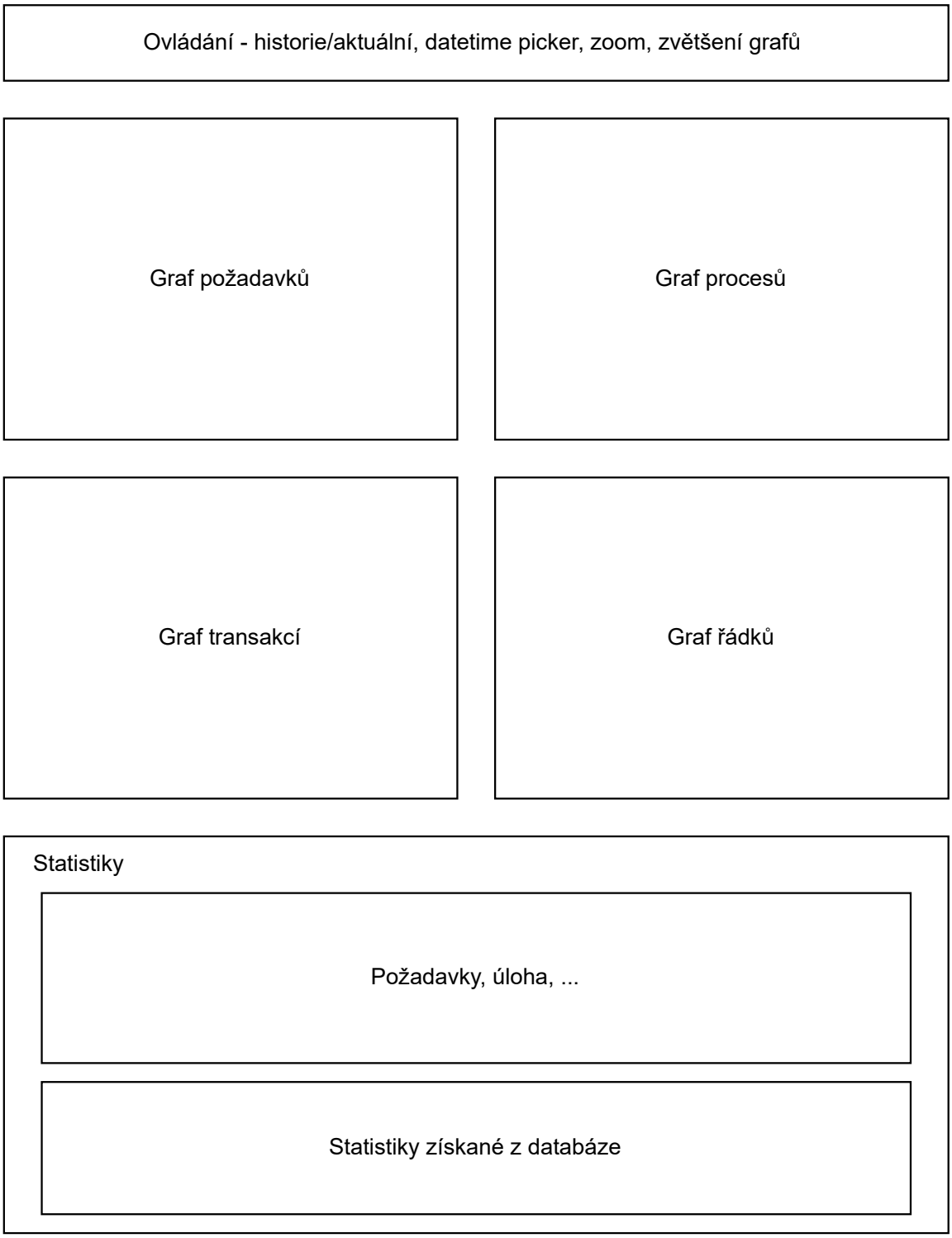
Na obrázcích A.1, A.2 a A.3 lze vidět náčrty rozhraní pro manuální řízení, řízení se scénářem a vizualizaci dat.



Obrázek A.1: Náčrt panelu manuálního řízení



Obrázek A.2: Náčrt panelu řízení podle scénáře

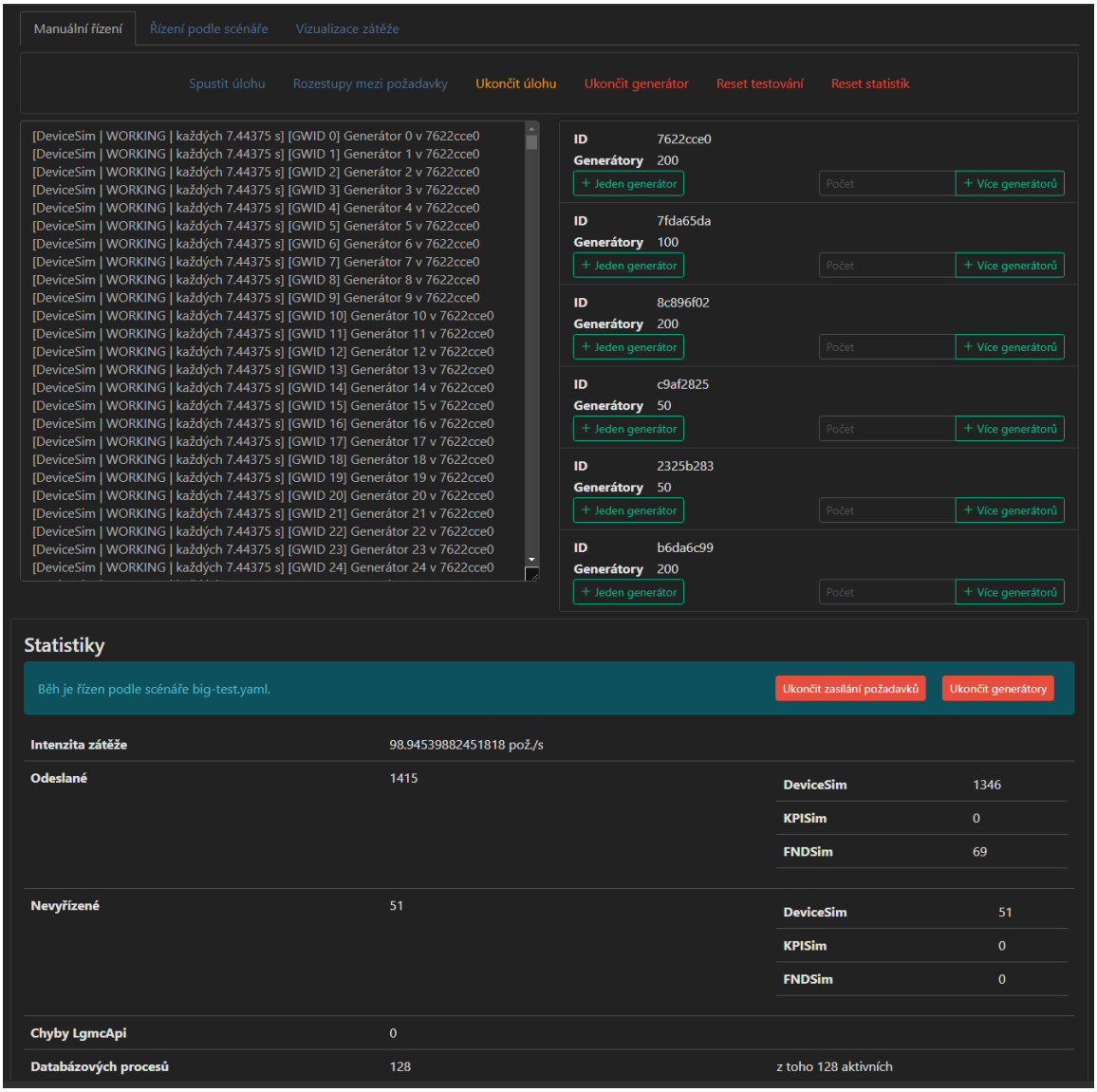


Obrázek A.3: Náčrt panelu vizualizace dat

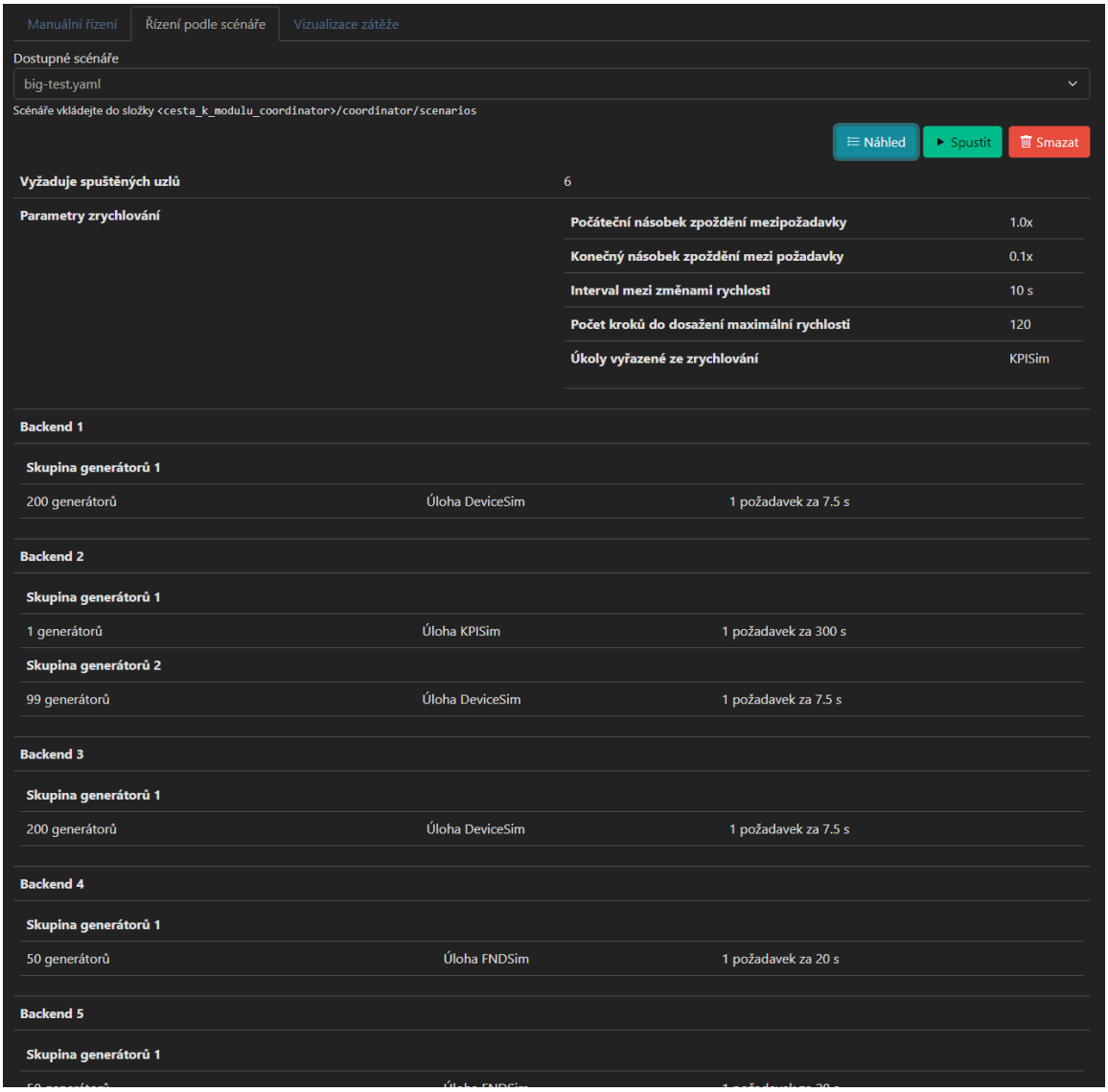
Příloha B

Snímky obrazovky realizace uživatelského rozhraní

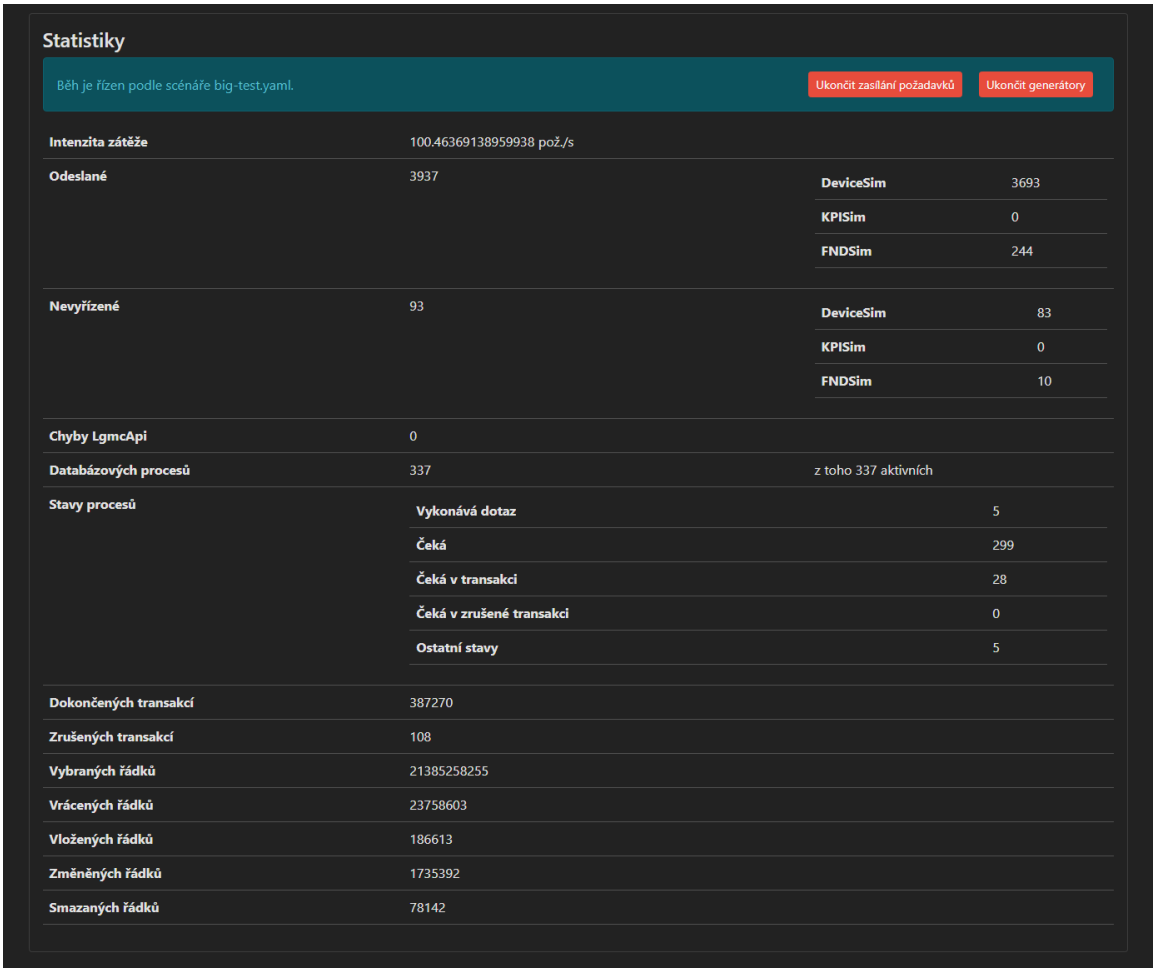
Obrázek B.1 zobrazuje snímek manuálního řízení testování, obrázek B.2 ukázku náhledu testovacího scénáře a obrázek B.3 ukazuje výpis automaticky aktualizovaných dat o testování.



Obrázek B.1: Screenshot záložky manuálního řízení



Obrázek B.2: Screenshot náhledu testovacího scénáře



Obrázek B.3: Screenshot výpisu sesbíraných dat