UNIVERZITA PARDUBICE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

DIPLOMOVÁ PRÁCE

2024 Bc. Silvia Čmilanská

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Distribuovaný systém pro geodetickou datovou kontrolu

Diplomová práce

2024 Bc. Silvia Čmilanská

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Silvia Čmilanská**
Osobní číslo:	**I22154**
Studijní program:	**N0613A140007 Informační technologie**
Téma práce:	**Distribuovaný systém pro geodetickou datovou kontrolu**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování

Cílem diplomové práce je vytvoření uceleného systému, který umožní efektivní a spolehlivou správu kontroly geodetických dat v distribuovaném prostředí. Celý systém bude postaven na platformě .NET s využitím kontejnerizace pomocí Dockeru. Komunikace bude zajištěna protokolem AMQP. Dále bude vytvořeno REST API pro předávání dat a poskytování jednotlivých operací v rámci distribuovaného systému. Součástí bude jednoduchý dotazovací formulář, který umožnění uživateli nahrát data a provést požadovanou kontrolu nad geodetickými daty.

Rozsah pracovní zprávy: **50**
Rozsah grafických prací:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

FREEMAN, Adam. Essential Docker for ASP.NET Core MVC. Imprint: Apress, 2017. ISBN 9781484227787.
NICKOLOFF, Jeff. Docker in Action. 1. United States of America: Manning Publications Co., 2016. ISBN 9781633430235.
RabbitMQ: Messaging that just works [online]. 2023 [cit. 2023-10-06]. Dostupné z:http://www.rabbitmq.com/semantics.html

Vedoucí diplomové práce: **Ing. Soňa Neradová, Ph.D.**
Katedra informačních technologií

Datum zadání diplomové práce: **8. listopadu 2023**
Termín odevzdání diplomové práce: **17. května 2024**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

prof. Ing. Antonín Kavička, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 30. listopadu 2023

Prohlašuji:

Práci s názvem Distribuovaný systém pro geodetickou datovou kontrolu jsem vypracovala samostatně. Veškeré literární prameny a informace, které jsem v práci využila, jsou uvedeny v seznamu použité literatury.

Byla jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019. Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 17. 5. 2024

Bc. Silvia Čmilanská

Poděkování

Tímto bych ráda poděkovala své vedoucí diplomové práce Ing. Soně Neradové, Ph.D. a mému vedoucímu v zaměstnání za poskytnutí námětu, ochotu a cenné rady při zpracování této práce. Dále bych chtěla poděkovat také své rodině a přátelům za pochopení a podporu v průběhu celého studia.

ANOTACE

Cílem diplomové práce bylo vytvoření systému pro efektivní a spolehlivou správu kontroly geodetických dat v distribuovaném prostředí. Distribuovaný systém je postaven na platformě .NET s využitím kontejnerizace pomocí Dockeru. Komunikaci zajišťuje protokol AMQP a jednotlivé aplikace jsou implementovány jako mikroslužby. Uživatelskou interakci se systémem poskytuje webová aplikace, která zobrazuje historii provedených kontrol, umožňuje nahrát data a provést požadovanou kontrolu nad geodetickými daty. Teoretická část práce se věnuje distribuovaným systémům, jejich architekturám a komunikačním modelům. Dále zmiňuje protokol AMQP, technologii Docker a XML formát. Praktická část poté popisuje návrh a implementaci jednotlivých součástí distribuovaného systému.

KLÍČOVÁ SLOVA

distribuovaný systém, AMQP, Rabbit MQ, .NET, Docker

TITLE

Distributed system for geodetic data control

ANNOTATION

The diploma thesis aims to create a system for effective and reliable management of geodetic control in a distributed environment. The distributed system is built on the .NET platform using containerization with Docker. Communication is provided by the AMQP protocol and individual applications are implemented as microservices. User interaction with the system is provided by a web application that displays the history of performed data controls. It also allows users to upload geodetic data and perform the required set of data controls. The theoretical part of the thesis is devoted to distributed systems, their architectures, and communication models. It also mentions the AMQP protocol, Docker technology, and XML format. The practical part then describes the design and implementation of individual components of the distributed system.

KEYWORDS

distributed system, AMQP, Rabbit MQ, .NET, Docker

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM ZKRATEK

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CLI	Command Line Interface
DS	Distribuovaný Systém
EBA	Event Based Architecture
GUID	Globally Unique Identifier
IDL	Interface Definition Language
IoT	Internet of Things
JVF DTM	Jednotný Výměnný Formát Digitální Mapy České Republiky
JWT	JSON Web Token
MPI	Message Passing Interface
OOA	Object-Oriented Architecture
P2P	Peer to Peer
REST	Representational State Transfer
RPC	Remote Procedure Call
SOA	Service-Oriented Architecture
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	eXtensible Markup Language
XSD	XML Schema Definition
YAML	YAML Ain't Markup Language

ÚVOD

Distribuované systémy představují v dnešní době digitalizace a internetu důležitou technologii umožňující efektivní zpracování a správu dat s využitím nezávislých zařízení, které mohou být umístěny na geograficky různých místech. Tato zařízení mezi sebou komunikují pomocí zasílání zpráv po síti. Distribuované systémy nabízejí výhody v podobě vysoké dostupnosti, odolnosti vůči selhání a škálovatelnosti, na rozdíl od klasických aplikací, které jsou navrženy jako centralizované systémy. Jednou z dalších důležitých vlastností distribuovaného systému je fakt, že uživatelům se jeví jako jeden celek, i když ve skutečnosti je implementován pomocí několika samostatných aplikací, které mohou běžet například i na různých zařízeních.

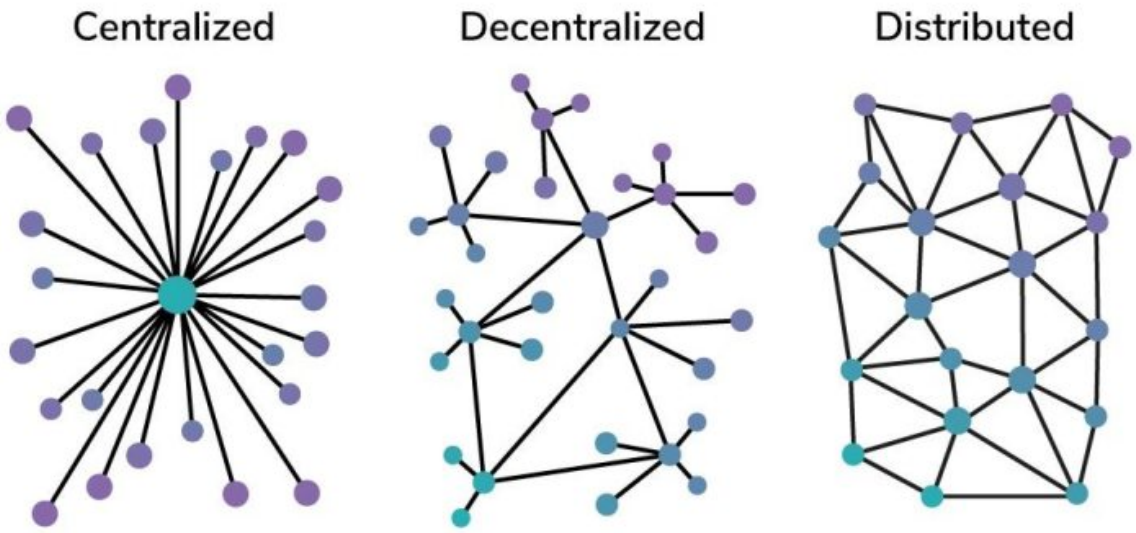
Hlavním cílem této diplomové práce bylo implementovat distribuovaný systém pro efektivní a spolehlivou správu kontroly geodetických dat, který je snadno škálovatelný, zvládá zpracovat mnoho uživatelských požadavků najednou a je odolný vůči chybám jednotlivých svých součástí. Aplikace v roli uzlů distribuovaného systému mají být implementovány na platformě .NET a celý systém má být nasazen pomocí Docker kontejnerů.

Teoretická část práce se zaměřuje na hlubší pochopení distribuovaných systémů a technologií využitých pro implementaci systému. Jsou zde popsány architektonické vzory a komunikační modely, které jsou nezbytné pro správné porozumění a návrh efektivního distribuovaného systému. Dále je v teoretické části kladen důraz na protokol AMQP a technologii Docker, jelikož hrají zásadní roli při implementaci komunikace a kontejnerizace služeb systému. Nakonec je popsán formát XML, který je používán pro výměnu dat mezi různými částmi systému.

Praktická část se věnuje návrhu řešení distribuovaného systému a poté popisuje konkrétní implementaci jednotlivých částí navrženého systému. Jsou zde podrobně popsány jednotlivé komponenty systému jako RabbitMQ zajišťující komunikaci, databáze pro uchování dat o kontrolách, služba Keycloak pro správu a ověřování uživatelů a samotné mikroslužby pro validaci XML dat, REST API a rozhraní kontrol. Nakonec se práce věnuje popisu uživatelského rozhraní webové aplikace poskytující možnost interakce se systémem.

1 DISTRIBUOVANÉ SYSTÉMY

Definice distribuovaných systémů je napříč literaturou velice rozmanitá. A. S. Tanenbaum například zobecněně charakterizuje distribuované systémy jako skupinu nezávislých výpočetních uzlů (počítačů), které se uživateli jeví jako jeden koherentní systém. Tyto výpočetní uzly fungují autonomně a komunikují mezi sebou pomocí zasílání zpráv po síti. Jednotlivé uzly tedy mohou být geograficky umístěny na různých místech aniž by koncový uživatel poznal při interakci se systémem rozdíl. [1]



Obrázek 1: Centralizovaný, decentralizovaný a distribuovaný systém. [4]

Na obrázku 1 je možné vidět vizualizaci rozdílů v topologii mezi centralizovanými, decentralizovanými a distribuovanými systémy. Zmíněné typy systémů mohou být jak malé velikosti o pouze několika zařízeních a uživatelích až po rozsáhlé systémy obsahující spolupracující zařízení napříč kontinenty. Jejich charakteristika, srovnání výhod a nevýhod například z hlediska problematiky škálovatelnosti a odolnosti proti chybám je uvedena v následujícím textu:

Centralizované systémy

Charakteristická je jedna centrální autorita - server ve funkci řídící jednotky celého systému. Spravuje data, komunikaci a řídí chování systému. To znamená, že jediné zařízení

provádí všechny výpočetní operace daného systému. Každý klient přistupující do systému tedy zasílá požadavek na daný centrální prvek. [3], [5]

- **Výhody:** Mezi výhody centralizovaných systémů patří jednodušší vývoj, správa zdrojů, aktualizací a zabezpečení, jelikož je vše soustředěno na jednom místě. Dále je toto řešení velice efektivní v případech, kdy je zapotřebí rychlého rozhodování nebo kontrola nad všemi operacemi a daty. [3], [5]
- **Nevýhody:** Důvodem proč se od tohoto řešení v dnešní době ustupuje jsou jeho slabé stránky. Jelikož je vše soustředěno do jednoho místa, systém představuje jediný bod selhání. Pokud nastane selhání serveru nebo chyba v aplikaci, ovlivněn bude celý systém a uživatelé ztratí přístup k datům či službě. V neposlední řadě jelikož jsou všechna data umístěna na jednom místě, představuje tato architektura bezpečnostní riziko. Dále také tato architektura umožňuje pouze velmi omezené možnosti pro škálování systému. [3], [5]

Decentralizované systémy

Na rozdíl od centralizovaného systému je zde centrální server nahrazen více uzly, které si mezi sebou rovnocenně rozdělují zpracování dat a řízení chodu systému pomocí síťové komunikace. Všechny tyto řídící uzly většinou uchovávají kopii prostředků, ke kterým uživatelé přistupují. Každé rozhodnutí je provedeno po souhlasu všech řídících uzlů, což zvyšuje transparentnost systému. Decentralizovaný systém je stejně náchylný k selhání jednotlivých uzlů jako centralizovaný systém. Výhodou této architektury je však, že pokud jeden nebo více uzlů selže, ostatní řídící uzly nadále běží a poskytují uživatelům data a přístup k požadované službě. [3], [5]

- **Výhody:** Decentralizované systémy zajišťují menší riziko celkového selhání systému díky odstranění problému jediného bodu selhání, jak již bylo popsáno výše. Dále mají lepší výkonnost, jelikož uživatel může přistupovat ke geograficky bližšímu uzlu s nižší přístupovou dobou než by byla ke vzdálenějšímu uzlu. [3], [5]
- **Nevýhody:** Tato architektura je mnohem složitější na správu na rozdíl od centralizovaného řešení, jelikož je zde potřeba koordinovat více uzlů pro udržování konzistence dat a řízení komunikace mezi nimi. Může vznikat problém pomalejší odezvy kvůli delší době zpracování rozhodování systému, na kterém se podílí více uzlů zároveň. Další problematikou je škálovatelnost, kde při potřebě zvýšit počet

uzlů nastává složitost implementace kvůli vzrůstající komplexnosti systému, kde je poté náročné udržet stejnou úroveň zabezpečení a decentralizace. [3], [5]

Distribuované systémy

Stejně jako u decentralizovaných systémů se ani zde nenachází centrální prvek, který by systém řídil. Avšak zde je zpracování operací a řízení systému rozděleno mezi všechny uzly systému, které jsou nezávislé. Každý uzel distribuovaného systému dokáže zpracovávat data a rozhodovat se naprosto nezávisle, ale ve vzájemné kolaboraci pro dosažení společného cíle. Zvenku přitom systém působí jako jeden celek. [3], [6]

- **Výhody:** Jelikož jsou distribuované uzly naprosto nezávislé, tak běh systému neovlivní ani selhání ostatních uzlů. Data jsou pro zlepšení dostupnosti redundantně ukládána v několika uzlech zároveň. Tato architektura poskytuje největší míru škálovatelnosti oproti ostatním typům systémů díky čemuž distribuované systémy zvládají zpracovávat opravdu velký provoz a objem dat. [3], [6]
- **Nevýhody:** Mezi nevýhody distribuovaných systémů patří velká komplexnost, díky které jsou náročné nejen na vývoj, správu a údržbu chodu celého systému, ale také na koordinaci komunikace mezi jednotlivými uzly. Z toho vyplývá, že jsou tyto systémy typicky dražší na vývoj a údržbu než centralizované nebo decentralizované systémy. [3], [6]

1.1 Hlavní vlastnosti

Pro hlubší vhled do způsobu návrhu distribuovaných systémů, jejich možností i kde je nutné činit kompromisy jsou zde popsány jejich charakterizující vlastnosti. Ne v každém případě je totiž vhodné systém implementovat jako distribuovaný. Podle Tanenbauma existují čtyři klíčové vlastnosti, které by distribuovaný systém měl nutně splňovat, aby jeho implementace měla význam. A to umožnění snadného přístupu ke zdrojům, skrytí distribuce zdrojů v síti, otevřenost systému a škálovatelnost systému. Tyto i ostatní charakteristiky jsou popsány níže. [1], [2]

1.1.1 Transparentnost

Jednou z klíčových charakteristik je transparentnost systému, jejímž cílem je skrytí skutečnosti, že uživatel interaguje s mnoha distribuovanými zdroji či zařízeními, které jsou propojeny sítí, a tudíž mohou být od sebe různé vzdálené. Zároveň tímto maskováním určitých funkcí distribuce usnadňuje transparentnost programátorům zjednodušení problematiky a možnost více se soustředit na návrh a implementaci konkrétní činnosti systému. Existuje několik forem transparentnosti, které se v distribuovaných systémech využívají. [1], [2] Dále je zmíněn výběr několika nejdůležitějších forem transparentnosti dle Tanenbauma:

- **Access** – Transparentnost přístupu je popisována jako skrývání rozdílů v reprezentaci dat a ve způsobu jak je ke zdrojům přistupováno. V praxi to znamená, že je použito stejné API pro lokální i vzdálený přistup. Pokud se tedy uživatel snaží získat určitý zdroj, měl by pro něj proces vypadat stejně v případě, kdy jsou data uložena lokálně nebo i když jsou data na vzdáleném serveru. Transparentnost uživatele odstíní od reálné architektury, vnitřní reprezentace dat daného operačního systému či souborového systému atd. na zařízení, které data poskytuje. [2], [10]
- **Location** – Lokační transparentnost je jednoduše skrytí skutečnosti, kde jsou data uložena. Aplikuje se výběrem takového logického pojmenování zdrojů, které neobsahuje žádnou část naznačující, jaké je fyzické umístění zdroje. Příkladem může být uniform resource locator neboli URL. Při přístupu ke zdroji na internetu pomocí URL z dané URL nelze vyčíst reálné umístění web serveru, na kterém je zdroj uložen. [2], [10]
- **Relocation + Migration** – Relokace skrývá uživateli skutečnost, že zdroj nebo procesy byly přisunuty na jinou lokaci - například do jiného datacentra. Migrace taktéž znamená přesunutí zdrojů na jinou lokaci, jedná se však o přesun během soustavného používání dané služby uživatelem aniž by došlo k jakémukoliv přerušení poskytování služby. [2], [10]
- **Replication** – V distribuovaných systémech bývají některé zdroje či procesy uchovávány nebo spouštěny v několika kopiích/instancích. Díky tomuto faktu je zajištěna mnohem vyšší dostupnost i výkonnost dané služby - například použití kopie ze serveru, který je geograficky bližší uživateli přistupujícímu k daným datům. Záro-

veň replikace služeb umožňuje nerušený běh a poskytování služeb v distribuovaném systému a i případě, že některý uzel selže. [10]

- **Concurrency** – Uživatel si není vědom, že zdroje ke kterým přistupuje mohou být sdíleny s několika dalšími konkurenčními uživateli. Například dva různí uživatelé mohou mít svá data uloženy ve stejné tabulce stejné databáze na stejném serveru. Do této kategorie spadá také paralelní zpracování procesů využívajících sdílené zdroje. Je však důležité aby souběžný přístup nenarušoval konzistenci dat. To je dosaženo pomocí zajištění výlučného přístupu ke zdrojům nebo transakčním zpracováním procesů. [10]
- **Failure** – Jednou z nejdůležitějších charakteristik pro uživatele je transparentnost selhání, díky které uživatel nezaznamená když některý z uzlů systému selže a následně je systémem obnoven do funkčního stavu. Služby jsou poskytovány bez přerušení. K selhání může dojít například z důvodu selhání služby, hardware nebo sítě. Skrývání selhání je jednou z nejvíce komplikovaných vlastností distribuovaných systémů. Zejména díky neschopnosti rozlišit mezi mrtvým uzlem, který selhal a velice pomalým uzlem například kvůli zahlcení sítě. [2], [10]

1.1.2 Škálovatelnost

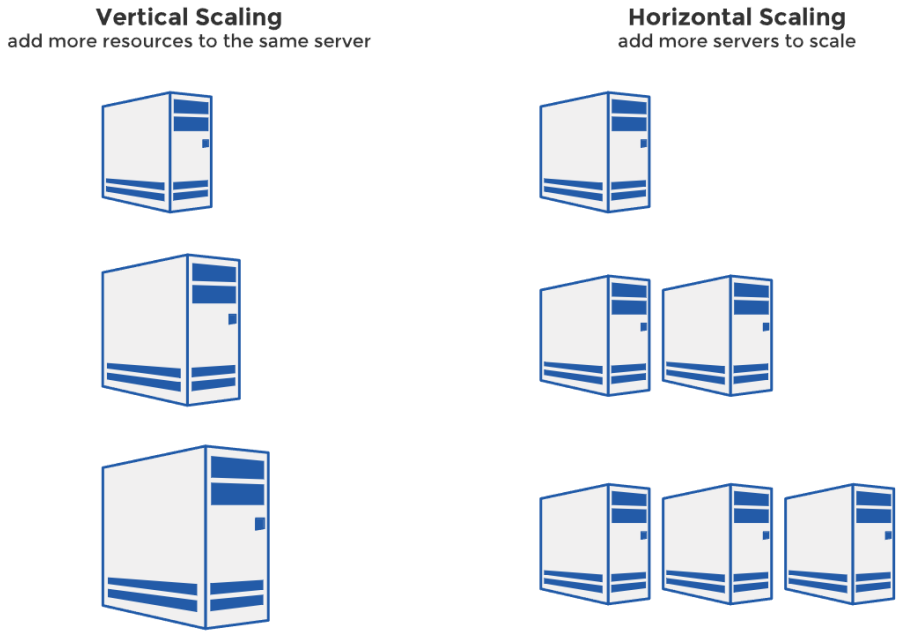
Další klíčovou vlastností distribuované architektury je hned vedle transparentnosti škálovatelnost. Škálovatelnost neboli rozšiřitelnost je schopnost systému reagovat na zvyšující se zátěž, nároky na výkon, množství zdrojů a množství uživatelů. Zajišťuje jednoduché přidělování nových prostředků, aby při zvýšených nárocích na systém nedošlo ke ztrátám výkonnosti či dostupnosti systému. [9]

Zátěž systému je měřitelná několika různými ukazateli. Například jako množství požadavků za jednotku času, počet čtení/zápisů, počet současně podporovaných uživatelů, frekvence datových záloh atp. [7]

Obecně rozlišujeme dva typy škálovatelnosti, a to horizontální a vertikální:

- **Vertikální škálovatelnost:** Jedná se o přidělení více zdrojů již existujícímu serveru. Například zvýšení výkonu procesoru, přidání paměti RAM nebo zvýšení kapacity úložiště atp. Tato metoda má však již své limity, kdy už nebude možné přidat více zdrojů danému serveru. Zároveň se s vertikálním škálováním pojí problém nedostupnosti serveru v průběhu instalace nových zdrojů. [8]

- **Horizontální škálovatelnost:** Ve chvíli kdy je dosaženo limitu vertikální škálovatelnosti jednoho zařízení, tak je řešením horizontální škálovatelnost, která přidává rozšíření v podobě dalšího serveru – neboli distribuci zdrojů mezi několik zařízení. Výhodou je, že provoz systému nebude při implementaci škálování přerušen, protože se jedná o pouhé přidání serveru do clusteru. Tato varianta je ovšem finančně nákladnější z hlediska pořizování nového hardware. [8]



Obrázek 2: Vertikální vs horizontální škálovatelnost. [7]

1.1.3 Otevřenost

Otevřený distribuovaný systém je takový systém, jehož komponenty lze snadno integrovat nebo používat v jiných systémech. Zároveň by měl sám umět využívat komponenty z jiných systémů. Proto, aby tyto požadavky byly splnitelné, bylo potřeba zavést standardizaci definice komponent, konkrétně syntaxe a sémantiky popisu jaké služby dané komponenty poskytují. Obecně rozšířený přístup je definice služeb pomocí rozhraní. Pro tyto účely byl standardizován jazyk Interface Definition Language (IDL). Pomocí IDL je pro služby možné popsat jejich metody s atributy, návratové hodnoty a případně výjimky. [2]

Aby byl distribuovaný systém otevřený, musí tedy splňovat tyto charakteristiky [9]:

- Rozhraní komponent by mělo být dobře a přesně definováno.
- Podoba rozhraní komponent by měla být standardizována.
- Integrace nových komponent k těm starým, nebo jejich nahrazení musí být snadná.

1.1.4 Sdílení zdrojů

Velice důležitým aspektem je umožnění uživatelům snadného přístupu i sdílení vzdálených zdrojů. Těmito sdílenými zdroji může být myšleno mnoho věcí, zejména se jedná například o úložiště, data, soubory, služby, periferní zařízení nebo sítě. [2]

1.1.5 Spolehlivost

Spolehlivost systému lze definovat jako míru toho, na kolik se můžeme spolehnout, že se systém bude chovat dle očekávání. Jak zde již bylo zmíněno, distribuované systémy jsou mnohem spolehlivější na rozdíl od centralizovaných řešení. Ovšem vyjádření spolehlivosti zde není tak jednoduché jako u centralizovaných řešení. Pokud centrální server centralizovaného systému selže, tak spadne celý systém. To se u distribuovaného systému nestane, jelikož systém pracuje na bázi nezávislých a autonomních uzlů. Tedy v případě, že jeden selže, ostatní nadále běží a systém může fungovat dál. Avšak problémem jsou částečná selhání, která mohou neočekávaně ovlivnit chod systému.

Cílem distribuovaného řešení je maskovat tato částečná selhání a opravovat je bez vědomí uživatele. Schopnost tolerance a maskování chyb je nazývána jako *fault tolerance*, neboli odolnost vůči chybám. [2]

Dle Tanenbauma lze spolehlivost distribuovaných systémů popsat pomocí těchto několika základních konceptů [2]:

- **Dostupnost** – poukazuje na připravenost systému k okamžitému používání s důrazem na pravděpodobnost, že systém bude v daném okamžiku plně funkční a schopen plnit požadované operace.
- **Spolehlivost** – soustředí se na zajištění nepřetržitého provozu systému v daném časovém intervalu. Vyjadřuje pravděpodobnost, že systém bude fungovat bez přerušení po stanovenou dobu.
- **Bezpečnost** – zaručuje, aby během dočasného výpadku běžného provozu systému nedošlo k žádným katastrofálním událostem. V real-time systémech jako například v systémech řízení jaderných reakcí nebo letecké dopravy musejí být dodržovány vysoké bezpečnostní standardy. Skutečně spolehlivý systém musí být zároveň bezpečným systémem. Mezi těmito dvěma vlastnostmi distribuovaných systémů je velice

úzká vazba. Bezpečnost je dosažena zejména omezením přístupu do systému pouze autorizovaným identitám a dodržováním integrity dat.

- **Udržovatelnost** – týká se toho, jak snadná nebo náročná bude oprava systému po jeho selhání včetně kapacity pro automatickou detekci a obnovu po poruše.

1.2 Architektura

Tato kapitola se zaměří na logickou organizaci distribuovaného systému do softwarových komponent, které společně tvoří jeho softwarovou architekturu. Správný návrh a implementace architektury systému je velice zásadní pro úspěšný vývoj velkých softwarových systémů.

Architektura DS je často formulována pomocí rozdělení systému do komponent, dále způsobem jakým komponenty mezi sebou komunikují, pomocí dat, které si mezi sebou komponenty předávají a také tím, jak jsou tyto komponenty pospojovány do uceleného pevně spolupracujícího systému pomocí konektorů.

Komponenta je modulární samostatnou jednotkou s dobře definovanými rozhraními, která je ve svém prostředí jednoduše vyměnitelná. Jak již bylo zmíněno v popisu základních vlastností distribuovaného systému – možnost nahradit komponentu, a to i za běhu systému, je velice zásadní vlastností, jelikož umožňuje opravení chyb bez nutnosti vypínat systém, takže nedochází k přerušení poskytování služeb.

Konektor je v architektuře distribuovaného systému obecně popisován jako mechanismus, který zprostředkovává komunikaci, koordinaci nebo spolupráci mezi komponentami. Jde o klíčový prvek, který umožňuje tok kontrolních signálů a dat mezi různými částmi systému. Může nabývat různých forem jako například prostředek pro vzdálená volání procedur, předávání zpráv nebo streamování dat.

Díky použití konektorů a komponent mohou být distribuované systémy konfigurovány do různých architektonických stylů mezi nimiž rozlišujeme dva hlavní typy, a to softwarové architektury a architektury na úrovni systému. V praxi jsou často tyto styly kombinovány pro dosažení optimální funkčnosti a efektivity systému. [2]

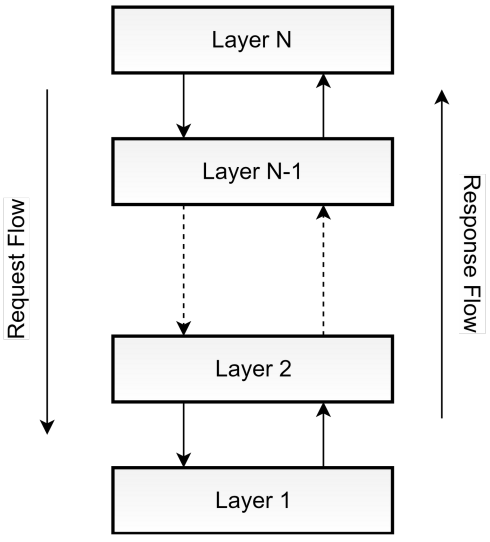
1.2.1 Softwarové architektury

Softwarová architektura DS pracuje s vysokoúrovňovými strukturami a návrhovými vzory, které určují organizaci softwarových komponent a jejich interakce v prostředí propojeném sítí. Hlavní důraz je kladen především na vnější, viditelné komponenty systému a jejich vzájemnou interakci. [11] Tato architektura zahrnuje několik běžně používaných architektonických stylů a může využívat hybridní přístupy, které kombinují různé elementy pro řešení složitosti reálných distribuovaných systémů. [2]

Vrstvená architektura

Vrstvená architektura, z angličtiny jako Layered Architecture, se vyznačuje svou modularitou, neboli schopností efektivně oddělovat jednotlivé komponenty do samostatných jednotek – vrstev. Komponenty jsou organizovány konkrétně do hierarchických vrstev, přičemž každá vrstva má specifickou odpovědnost a funkci v rámci celkového systému. [12]

Komunikace mezi vrstvami probíhá sekvenčně od vrchních vrstev k dolním a naopak, což umožňuje systematické zpracování požadavků a odpovědí. Spodní vrstvy poskytují služby těm vrchním. Jak je zřejmé z obrázku 3, požadavky proudí z vrchních vrstev k těm spodním. Odpovědi jsou následně posílány opačným směrem zdola nahoru. [13]



Obrázek 3: Vrstvená architektura [13]

V některých případech je však výhodné implementovat komunikaci pomocí koordinace napříč vrstvami neboli cross-layer coordination. Pomocí této koordinace je možné přesko-

čit jakoukoliv sousední vrstvu, což poskytuje lepší výsledky z hlediska zvýšení výkonu systému. [13]

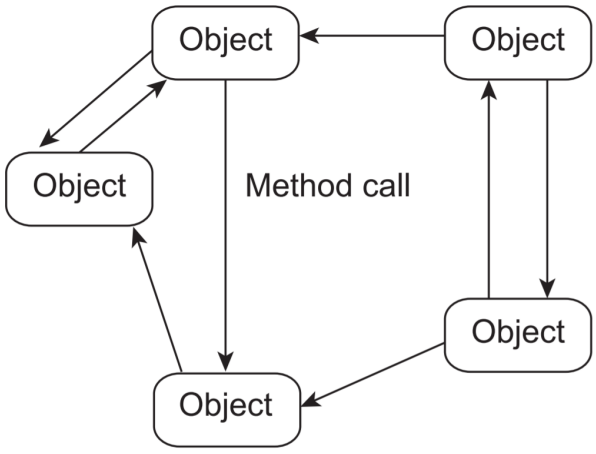
Ačkoliv je tento architektonický styl velice oblíbený, jednou z jeho hlavních nevýhod je často silná závislost mezi jednotlivými vrstvami. Dále hrozí vznik úzkého hrdla při komunikaci mezi vrstvami, pokud je některá z vrstev pomalejší při zpracovávání požadavků než jsou ostatní. [2]

Objektově řízená architektura

Objektově řízená architektura je založená na uspořádání volně propojených objektů. Na rozdíl od vrstvené architektury zde není žádné pevné uspořádání jako jsou vrstvy, ani žádná předem stanovená sekvence kroků.

Každá komponenta je reprezentována jako objekt. Jednou z klíčových vlastností této architektury je abstrakce dat. Podobně jako u objektově orientované architektury, objektově řízená architektura zapouzdřuje data do objektů a poskytuje rozhraní pro přístup k zapouzdřeným datům a manipulaci s nimi. Díky této abstrakci jsou interní data chráněna před vnějšími komponentami, tudíž abstrakce podporuje konzistenci a integritu dat. Na rozdíl od OOA tato architektura zapouzdřuje pouze data, ne chování jako jsou funkce či metody. [15]

Interakce mezi všemi objekty jsou provozovány pomocí konektorů nebo rozhraní, typicky pomocí přímého volání metod. Mezi způsoby komunikace patří RPC – remote procedure call neboli vzdálené procedurální volání jako například Java RMI, webové služby nebo volání REST API. [13]



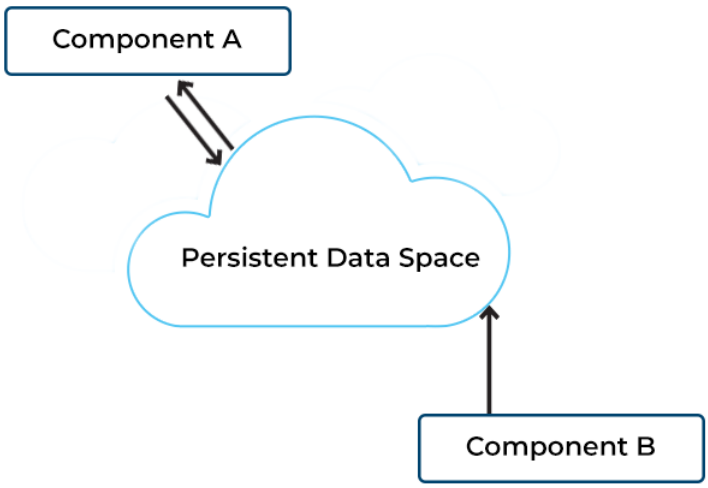
Obrázek 4: Objektově řízená architektura [13]

Datově centrická architektura

Datově centrická architektura distribuovaných systémů je postavena na ústředním úložišti dat, pomocí kterého probíhá primární komunikace mezi komponentami systému. Toto centrální úložiště může být aktivní nebo pasivní a slouží jako společná databáze pro všechny produkující (producenty) a konzumující (konzumenty) části systému. [14]

Jádrem architektury je centrální datové úložiště, které může být realizováno například jako SQL databáze. Ta poté slouží pro uchování všech dat týkajících se uzlů systému. Veškerá interakce mezi komponentami systému probíhá prostřednictvím daného centrálního datového úložiště, což zajišťuje standardizovaný přístup k datům. Producenti vytvářejí a ukládají data do společného centrálního úložiště a konzumenti si z něj mohou data vyžádat. [13] Úložiště dat bývá často přístupné prostřednictvím dobře definovaných API, což usnadňuje integraci a výměnu dat s externími systémy nebo službami.

Díky centralizaci správy dat a poskytování standardizovaných přístupových bodů datově centrická architektura zajišťuje, že data zůstávají konzistentní napříč celým systémem. Architektura je ideální pro systémy, kde jsou data hlavním aktivem, jako jsou databáze, informační systémy a datové sklady. [15]



Obrázek 5: Datově centrická architektura [14]

Událostmi řízená / Publish-Subscribe architektura

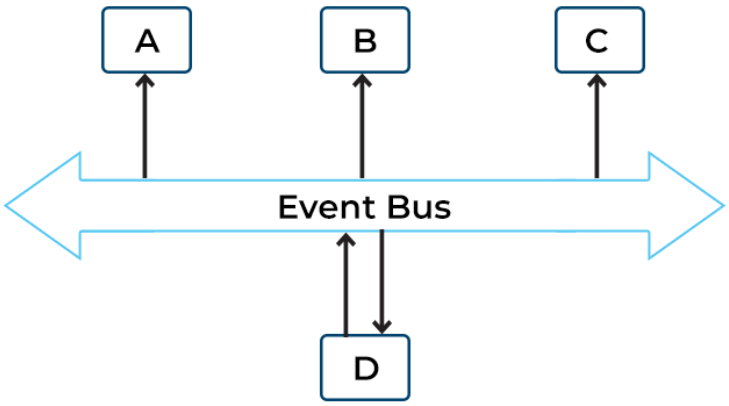
Hlavní charakteristikou událostmi řízené architektury, z angličtiny Event-Based Architecture (EBA), je že veškerá komunikace probíhá pomocí generování, detekce a zpracování

událostí (eventů) v systému. Eventy představují významné události nebo změny stavu, pomocí kterých komponenty v systému nepřímo komunikují. [15]

Tento architektonický styl je založen na publish-subscribe architektuře. Obsahuje tyto klíčové komponenty [15]:

- **Producenti událostí** – Komponenty zodpovědné za generování událostí. Odesílají události ve chvíli kdy jsou splněny specifické podmínky nebo provedeny dané akce. Může se jednat například o uživatelská rozhraní, senzory nebo externí systémy.
- **Konzumenti událostí** – Komponenty, které se přihlásí k odběru konkrétních typů událostí a reagují na ně předdefinovanými akcemi. Konzumenti mohou být například různé softwarové komponenty, služby nebo externí systémy, které interagují se systémem řízeným událostmi.
- **Broker událostí / sběrnice událostí** – Prostředník mezi producenty a spotřebiteli událostí, zajišťující směrování událostí správným spotřebitelům na základě jejich přihlášení k odběru. Mezi typické brokery událostí patří fronty zpráv, publish-subscribe systémy a událostmi řízený middleware.

Komunikace probíhá následovně. Ve chvíli kdy je vygenerována událost, pošle se do sběrnice událostí (event bus). Všechny komponenty, které odebírají daný typ události jsou upozorněni, že k události došlo. Pokud má některá komponenta o událost zájem, může se k ní přihlásit a získat ji ze sběrnice. Konzument tak získá přístup ke všem informacím, které jsou v události obsaženy, a podle toho je zpracovat. [13] Tato architektura je obzvláště vhodná pro vývoj reaktivních, realtimových a volně propojených distribuovaných systémů. [15]



Obrázek 6: Událostmi řízená architektura [14]

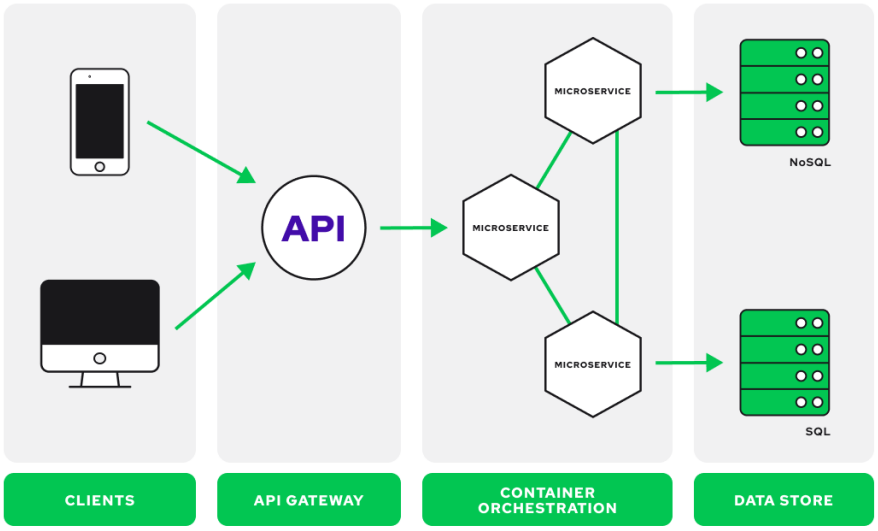
Architektura orientovaná na služby

Servisně orientovaná architektura (SOA) je předchůdcem mikroslužeb a zaměřuje se na integraci a poskytování služeb na úrovni aplikací nebo celých podnikových divizí. Na rozdíl od mikroslužeb, které se koncentrují na izolované funkční celky, SOA obaluje širší rozsah funkcionalit, často zahrnující celé aplikační nebo databázové systémy v rámci jednotlivých servisních uzlů. Komunikace mezi službami probíhá prostřednictvím zpráv přes síťové rozhraní.

Přestože mikroslužby jsou výhodnější z hlediska větší rozložitelnosti a škálovatelnosti systému, SOA zůstává silnou volbou pro scénáře, kde je požadována integrace rozsáhlých systémových funkcionalit. [16], [17]

Mikroslužby

Architektura mikroslužeb, nebo také mikroservisní architektura, je moderní přístup k vývoji softwarových aplikací, který je založen na principu rozdělení aplikace na soubor malých, nezávislých služeb. Každá mikroslužba je specializovaná na konkrétní úkol nebo funkci a komunikuje s ostatními službami prostřednictvím dobře definovaných API. Tento styl architektury vznik na základně servisně orientované architektury (SOA) a řeší její nedostatky. Umožňuje autonomní provoz jednotlivých komponent, což znamená, že každá mikroslužba může být vyvíjena, nasazována, škálována a provozována nezávisle na ostatních, aniž by docházelo k jejich vzájemnému ovlivňování. [18]



Obrázek 7: Architektura mikroslužeb [19]

Modularita mikroslužeb umožňuje, aby jednotlivé služby mohly využívat rozdílné technologie pro ukládání a zpracování dat a mohly být psány v různých programovacích jazycích, což představuje značnou výhodu oproti monolitickým architekturám, kde všechny procesy na sobě úzce závisí, jelikož jsou provozovány jako jediná služba.

Jednou z klíčových vlastností mikroservisní architektury je její schopnost decentralizace. Ta je dosažena například použitím nezávislých databází pro každou službu, což zajišťuje volnou vazbu a zlepšuje konzistenci a odolnost systému. Dalším důležitým prvkem architektury je API Gateway, která funguje jako centrální bod pro všechny klientské požadavky, směruje je k příslušným mikroslužbám a zjednodušuje tak komunikaci v rámci systému. Z hlediska podpory asynchronní komunikace je výhodné implementovat také frontu zpráv. Obecně komunikace mezi mikroslužbami probíhá prostřednictvím kombinace rozhraní REST API, streamování událostí a zprostředkovatelů zpráv jako jsou message brokery a již dříve zmíněné fronty zpráv.

Ačkoliv má tato architektura mnoho výhod, její implementace přináší i mnoho výzev. Jelikož je architektonicky rozdělena na mnoho částí, testování a detekce chyb jsou komplexní a mnohem složitější úkony než u monolitické aplikace. Dále také vyšší režie spojená s koordinací mezi jednotlivými mikroslužbami může vést ke snížení výkonu, tím pádem tato architektura nemusí být vhodnou volbou pro aplikace vyžadující vysoký výkon a nízkou latenci. [20]

Hybridní architektura

V rámci návrhu komplexního distribuovaného systému není snadné najít univerzální řešení, které by dokonale vyhovovalo všem požadavkům. Každý architektonický styl má své výhody i nevýhody. Hybridní architektura DS nabízí řešení v podobě kombinace několika architektonických stylů pro dosažení optimálního výkonu a funkčnosti systému.

Cílem je vytvořit řešení, které nejlépe odpovídá konkrétním potřebám aplikace nebo systému tím, že jsou kombinovány nejlepší aspekty z několika různých vybraných architektur. Tato strategie umožňuje vytvářet systémy, které jsou nejen efektivnější a škálovatelnější, ale také bezpečnější a snadněji udržovatelné. [15]

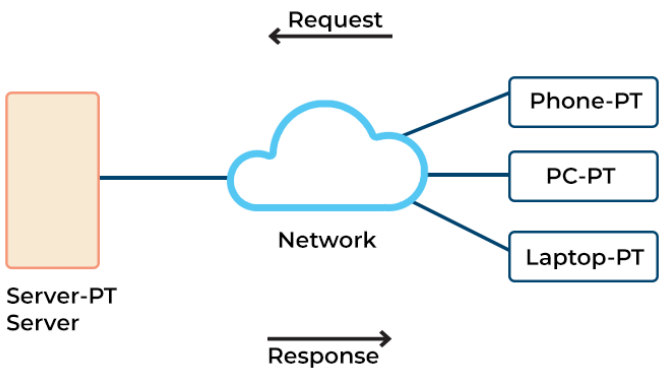
1.2.2 Systémové architektury

Systémová architektura, často označovaná jako architektura na systémové úrovni nebo vysokoúrovňová architektura, definuje převládající strukturu, komponenty a interakce komplexního systému. Zabývá se nejen softwarovými, ale i hardwarovými aspekty a zajišťuje, že všechny podsystémy spolupracují, aby dosáhly zamýšlené funkčnosti a cílů systému. [15]

Klient-Server

Dle zdrojů [15] a [18]. Architektura klient-server je jednou ze základních architektur distribuovaných systémů. Skládá se ze dvou hlavních komponent: klient a server.

- **Klient** – typicky zařízení (PC, notebook, mobilní telefon, IoT zařízení, atd.) nebo aplikace, díky které je uživateli umožněna interakce se systémem pro vyžádání služby nebo dat ze serveru. Prezentuje uživatelské rozhraní a zpracovává uživatelské vstupy. Může se jednat jak o webovou aplikaci, tak o desktopovou aplikaci.
- **Server** – může se jednat o server fyzický, virtualizovaný nebo založený na cloudu. Jsou to typicky zařízení vysokého výkonu s velkým úložištěm a instalovanou robustní pamětí aby byly schopny zpracovávat mnohonásobné požadavky od různých klientů současně. Zastává klíčovou roli zpracování požadavků klientů, provádění požadovaných akcí a správy dat. Jeho práce probíhá na pozadí, kde neustále čeká na příchozí požadavky a reaguje na ně. Server může nabývat různých forem, včetně webových serverů, databázových serverů, emailových serverů a dalších, přičemž každý typ serveru je specializovaný na určité úkoly. Tato specializace umožňuje efektivní a cílené zpracování různorodých požadavků a správu dat, což je zásadní pro hladký chod a výkonnost celého distribuovaného systému.



Obrázek 8: Kient-Server architektura [14]

Skutečně distribuovaný klient-server systém zahrnuje více serverových uzlů pro distribuci klientských připojení, čímž je zabráněno degradaci struktury na centralizovanou. Většina moderních architektur klient-server jsou klienti, kteří se připojují k zapouzdřenému distribuovanému systému na serveru. [18]

Jednou z klíčových vlastností architektury klient-server je existence centralizované bezpečnostní databáze, která obsahuje bezpečnostní údaje, jako jsou přihlašovací údaje a podrobnosti o přístupu. Uživatelé se nemohou přihlásit k serveru bez ověření přístupu. Tato vlastnost přináší architektuře klient-server vyšší míru stability a bezpečnosti ve srovnání s peer-to-peer systémy. [13]

Výzvou architektury klient-server je zvýšená složitost a potenciální úzká místa ve škálovatelnosti a zvýšený síťový provoz, které vyžadují pečlivé plánování a správu. Přesto architektura klient-server zůstává preferovaným modelem pro řadu aplikací, včetně webových aplikací, e-mailových služeb a databázových systémů, díky její schopnosti centralizovaného zabezpečení, škálovatelnosti prostřednictvím vyvažování zátěže a snadnější údržby. [16], [18]

N-tier (Multitier)

Multi-tier (n-tier) architektura, zahrnující i zmíněnou třívrstvou architekturu, je koncept, který byl poprvé využit v rámci podnikových webových služeb. Tato architektura dělí funkce aplikace do fyzicky oddělených vrstev. Díky této separaci mohou vývojáři upravovat nebo přidávat specifické vrstvy bez nutnosti modifikovat celou aplikaci, což zvyšuje její flexibilitu, znovupoužitelnost a škálovatelnost. [18]

Tento koncept je založen na klient-sever architektuře, kde je server rozdělen do n dalších uzlů, díky čemuž je docíleno rozdělení odpovědností serveru. Například některé uzly mohou asynchronně zpracovávat dlouhodobé úlohy a ostatní uzly serveru jsou poté volné pro zpracování požadavků uživatelů. [16]

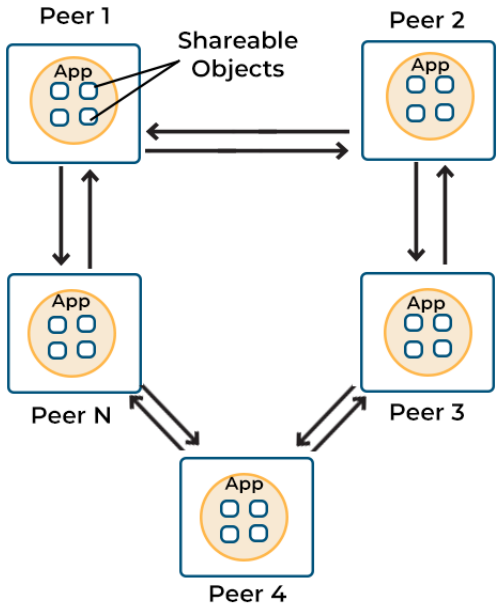
Nejčastěji používaným příkladem multi-tier architektury je právě třívrstvá architektura (thee-tier), která se skládá z prezentační vrstvy, logické vrstvy a datové vrstvy, a může být provozována na oddělených procesorech.

Komponenty třívrstvé architektury [18]:

- **Prezentační vrstva (Presentation Tier)** – vrstva tvořící uživatelské rozhraní architektury. Její hlavní úlohou je zpracování uživatelské interakce a zobrazování dat.
- **Logická vrstva (Logic Tier)** – známá také jako aplikační vrstva nebo vrstva obchodní logiky. Je zodpovědna za zpracování funkcionality systému. Provádí příkazy, logická rozhodnutí a výpočty.
- **Datová vrstva (Data Tier)** – vrstva, kam jsou ukládána data a odkud jsou také data načítána. Zahrnuje mechanismy pro ukládání dat, jako jsou databázové servery a sdílené soubory, a poskytuje API pro správu uložených dat.

Peer-to-Peer

Architektura peer-to-peer (P2P) představuje decentralizovaný a distribuovaný model, který je zásadní pro komunikaci v síti a sdílení dat. Na rozdíl od tradičních klient-server modelů, kde jsou požadavky a zdroje zpracovávány centrálním serverem, P2P sítě umožňují jednotlivým uzlům (peerům) fungovat jak v rolik klienta, tak i v roli serveru. Ve chvíli, kdy uzel požaduje službu, chová se jako klient. Pokud poskytuje službu, je považován za sever. Z daných důvodů tato architektura dobře splňuje distribuovaný a kolaborativní přístup. [14], [18]



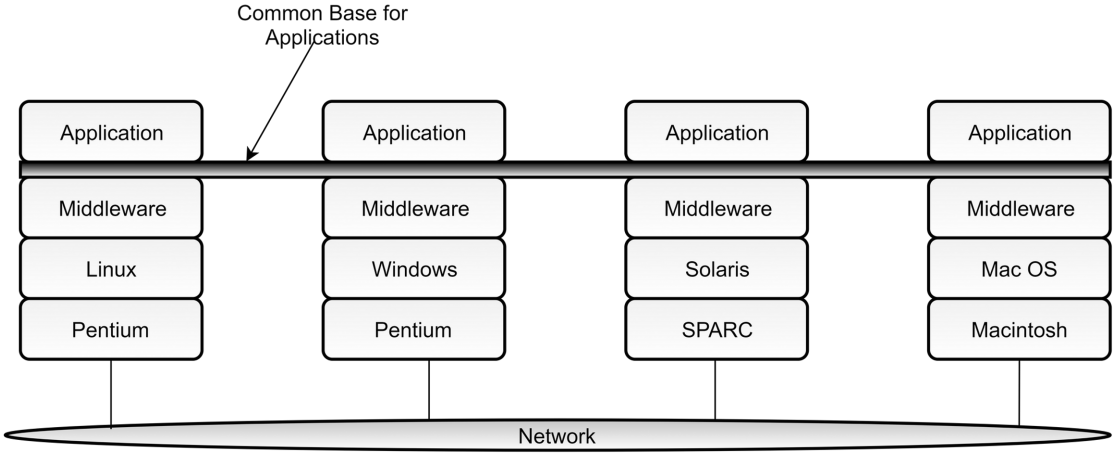
Obrázek 9: Peer-to-Peer architektura [14]

Významnou vlastností P2P systému je redundance uzlů. Každý uzel nese plnou instanci aplikace obsahující jak prezentační, tak datové vrstvy. Když je do systému zaveden nový peer, objeví a propojí se s ostatními peery, aby synchronizoval svůj lokální stav s celkovým systémem. Tato redundance zajišťuje odolnost systému, kdy selhání jednoho uzlu minimálně ovlivní celkovou síť.

Další klíčovou výhodou P2P systému je zvýšení kapacity systémových zdrojů s každým dalším připojeným uzlem. Čím více je do systému připojeno uzlů systém disponuje větším množstvím zdrojů z hlediska například šířky pásma, výpočetního výkonu či datového úložiště. [18]

1.2.3 Middleware v distribuovaných systémech

Middleware v distribuovaných systémech funguje jako most poskytující společnou platformu pro různý hardware, operační systémy a aplikace, které chceme nasadit v distribuovaném prostředí. Díky této společné vrstvě umožňuje jednotnou komunikaci a interakci mezi různými komponentami systému. Dále také nabízí služby, které přesahují základní funkčnost operačních systémů, a umožňuje tím lepší integraci a funkčnost komponent distribuovaného systému. [2], [13]



Obrázek 10: Middleware [13]

Klíčovými prvky middleware distribuovaných systémů jsou dva návrhové vzory, a to wrappery a interceptory.

Wrappery

Wrappery slouží k odstranění problémů s rozdíly mezi nekompatibilními rozhraními různých komponent distribuovaného systému. Wrapper, jiným názvem adapter, vytváří rozhraní přijatelné pro klienta tím, že transformuje jeho funkce do funkcí, které jsou dostupné v komponentě DS. Příkladem může být situace, kdy je zapotřebí integrovat existující komponenty s novým systémem a rozhraní těchto starších komponent se ukáží jako nevhodná pro všechny aplikace.

V kontextu distribuovaných systémů mohou wrappery sloužit jako rozhraní pro vzdálené objekty nebo jako adaptéry pro komunikaci s externími službami. Tím se řeší problém nekompatibilních rozhraní a zároveň se rozšiřuje možnost využití existujících systémových komponent.

Z popisu je zřejmé, že dané řešení není jednoduše škálovatelné, protože každá aplikace by musela mít vlastní wrapper pro každou aplikaci, se kterou chce komunikovat. Proto ve snaze snížit počet potřebných wrapperů v distribuovaných systémech je často využíván middleware společně s konceptem tzv. brokera. Broker představuje logicky centralizovanou komponentu, která řídí veškerý přístup mezi různými aplikacemi. Typickým příkladem je message broker, který umožňuje aplikacím odesílat požadavky obsahující informace o potřebných datech nebo službách. Broker, disponující znalostmi o všech relevantních aplikacích, poté kontaktuje odpovídající aplikace, může kombinovat a transformovat jejich odpovědi a výsledek vrátí zpět iniciující aplikaci. [2]

Interceptory

Interceptory představují softwarové konstrukty, které umožňují přerušit standardní průběh vykonávání operací v systému (například volání metod) a zasahovat do něj pomocí spuštění specifického kódu aplikace. Interceptory jsou klíčové pro přizpůsobení middleware konkrétním potřebám aplikace, což je nezbytné pro dosažení otevřenosti a flexibility middleware. Mohou například umožnit, aby volání metody objektu A na objekt B (který se může nacházet na jiném stroji) bylo transformováno do obecného objektového volání prostřednictvím middleware. To je obzvláště užitečné v situacích, kdy je objekt B replikován, a interceptor může zajistit, že volání je provedeno pro každou repliku. Tato flexibilita umožňuje middleware lépe reagovat na dynamické změny v distribuovaném prostředí. [2]

1.3 Komunikační modely

Základem všech distribuovaných systémů je komunikace mezi procesy a uzly systému, díky níž si mohou mezi sebou vyměňovat informace. Komunikační modely hrají zásadní roli v tom, aby systém fungoval správně a efektivně. Tato kapitola se zaměřuje na základní pravidla a protokoly, které umožňují procesům komunikovat.

Dle Tanenbauma lze komunikaci v distribuovaných systémech klasifikovat do čtyř kategorií, a to persistentní, transientní, synchronní a asynchronní komunikace. V praxi jsou v jednotlivých komunikačních modelech kombinovány různé typy persistence a synchronizace. [2]

- **Persistentní komunikace** – odeslaná zpráva je uložena v middleware tak dlouho, dokud není doručena příjemci. Díky této persistenci je možné, aby odesílatel ukončil aplikaci hned po odeslání zprávy, nebo aby příjemce v čase odeslání nebyl aktivní. I ve zmíněných případech bude zpráva korektně doručena.
- **Transientní komunikace** – zpráva je v systému uchována pouze po dobu kdy jsou aktivní jak odesílající aplikace, tak aplikace co zprávu přijímá. V případě, že middleware nemůže zprávu doručit kvůli přerušení přenosu nebo neaktivitě příjemce, zpráva je zahozena.
- **Asynchronní komunikace** – umožňuje odesílateli pokračovat ihned po odeslání zprávy. Zpráva je dočasně uložena middleware hned po odeslání.
- **Synchronní komunikace** – blokuje odesílatele, dokud není jeho požadavek přijat. Synchronizace může nastat v několika bodech: když middleware převezme požadavek k přenosu, když je požadavek doručen příjemci, nebo když je požadavek plně zpracován a příjemce vrátí odpověď.

1.3.1 Request-Response

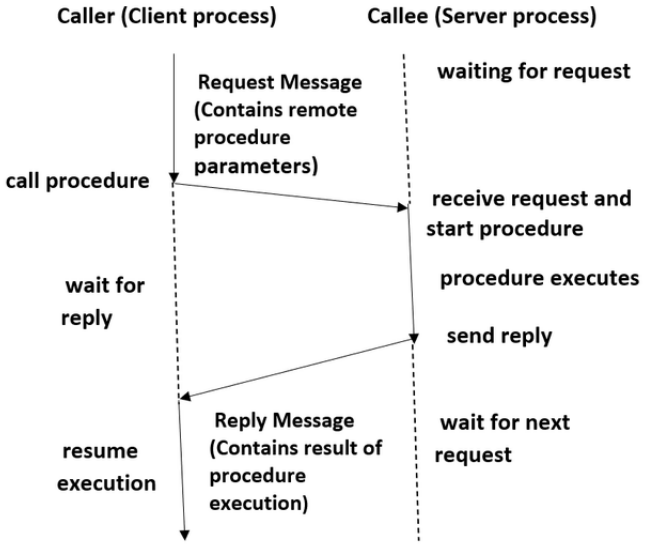
Model request-response je komunikační paradigma, ve kterém jeden uzel (klient) odesílá požadavek jinému uzlu (serveru) a čeká na odpověď. Jedná se o synchronní komunikaci, kde klient iniciuje akci, server zpracuje požadavek a vrátí odpověď. Dokud klient nedostane zpět odpověď tak čeká.

Tento model je široce využíván v architekturách klient-server, jako jsou webové prohlížeče, které žádají o webové stránky. Jeho hlavními výhodami jsou jednoduchost, snadná implementace a přímá kontrola nad komunikací.

Přestože je tento model snadný na implementaci a správu, tak má značné nevýhody ve svém blokujícím chování kvůli synchronní povaze komunikace. Tím dochází k obsazování zdrojů a snížení reakční schopnosti systému, což může být v některých aplikacích limitujícím faktorem. [12]

1.3.2 RPC

Remote Procedure Call (RPC) je komunikační technologie, která umožňuje aplikaci volat procedury umístěné na jiném zařízení stejně jako kdyby byly lokální. Tato metoda implementuje jednu z hlavních vlastností distribuovaných systémů tím, že skrývá složitost posílání zpráv před programátorem, čímž se snaží o dosažení přístupové transparentnosti. Díky tomu uživatel nebo aplikace nemají tušení, že volaná funkce probíhá na jiném stroji. [2], [21]



Obrázek 11: Průběh Remote Procedure Call [21]

Je nutné, aby volající a volaná strana souhlasily s formátem zpráv, které si vyměňují, a dodržovaly stejný protokol pro přenos složitých datových struktur. Pro sjednocení komunikace a posílaných dat je na straně klienta i serveru pomocná struktura, sloužící jako zprostředkovatel mezi klientem a serverem. Jedná se o serverový a klientský stub.

Klientský stub serializuje odesílané parametry do vhodného formátu pro síťový přenost a deserializuje odpoveď serveru. Serverový stub působí na straně serveru a prování opačný proces. Průběh RPC komunikace je zřejmý z obrázku 11. [2], [22]

1.3.3 Message-Oriented

Komunikační schéma orientované na zprávy je velmi důležitou alternativou k synchronní komunikaci typu RPC. RPC komunikace díky své synchronní povaze blokuje klienta dokud není jeho požadavek zpracován. Zatímco komunikace orientovaná na zprávy pracuje asynchronně, a proto je zvláště užitečná v situacích, kdy nelze předpokládat, že přijímající strana je v době odeslání požadavku aktivní. Rozlišujeme transientní a persistentní komunikaci orientovanou na zprávy. [2]

Transientní zasílání zpráv

Transientní zasílání zpráv v distribuovaných systémech zahrnuje komunikační technologie Berkeley Socket Primitives a MPI, které poskytují různé modely pro zpracování a přenos zpráv bez ukládání na disk. Díky této vlastnosti jsou tyto modely zaměřeny na využití pro rychlou a dočasnou komunikaci. [2]

- **Sockety** – poskytují jednoduchý způsob orientovaného zasílání zpráv nad transportní vrstvou. Sockety slouží jako abstraktní koncové body pro zápis a čtení dat a jsou základem pro navázání spojení mezi aplikacemi na různých strojích. Typické operace zahrnují vytváření socketu, přiřazení lokální adresy (bind), naslouchání na socketu (listen) a přijímání spojení (accept). [2], [23]
- **MPI** – Message-Passing Interface je pokročilejší systém pro předávání zpráv v rámci paralelních a distribuovaných výpočtů. Interface umožňuje flexibilní volání rutin z jazyků C, C++, Fortran, C#, Java nebo Python a je optimalizován pro hardware, na kterém běží. MPI poskytuje výkonnost a portabilitu a je považován za průmyslový standard pro paralelní programování. [24], [25]

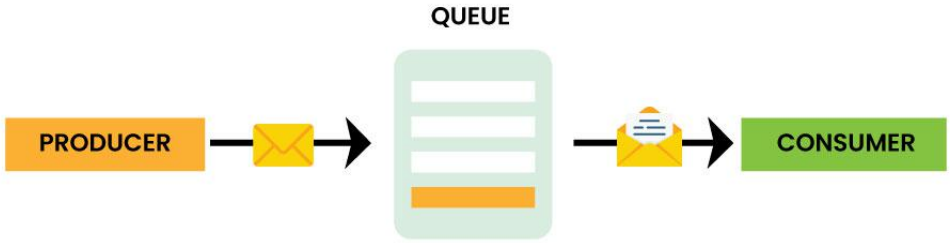
Persistentní zasílání zpráv

Do kategorie persistentního zasílání zpráv v distribuovanch systémech spadají komunikační modely front zpráv neboli Message-Oriented-Middleware, které poskytují rozsáhlou podporu pro persistentní asynchronní komunikaci. Na rozdíl od systémů jako Berkeley

sockety nebo MPI, které jsou zaměřeny na rychlý přenos zpráv, jsou systémy front zpráv navrženy pro zpracování přenosů, které mohou trvat delší dobu. Díky uchování zpráv v mezičase mezi odesláním a přijetím mohou fungovat bez potřeby aktivní účasti odesílatele nebo příjemce. [23]

Fronty zpráv

Komunikační model front zpráv představuje efektivní, spolehlivý a škálovatelný způsob pro výměnu dat mezi nezávislými komponentami distribuovaného systému. Aplikace mezi sebou komunikují vkládáním zpráv do specifických front na základě publish-subscribe komunikačního modelu. Tyto zprávy jsou posléze předávány mezi komunikačními servery (na kterých běží broker nebo správce front) a doručovány cílové aplikaci, i kdyby byla v době odeslání zprávy nedostupná. Každá aplikace může mít svou vlastní frontu, do které ostatní aplikace mohou zasílat zprávy. Fronta může být čtena buď jednou přiřazenou aplikací, nebo může být i sdílena mezi více vybraných aplikací. [2]



Obrázek 12: Komunikace pomocí front zpráv [26]

Důležitým aspektem systémů front zpráv je, že odesílateli je poskytnuta pouze záruka, že jeho zpráva bude nakonec vložena do fronty příjemce. Poté však už není nijak zaručeno, kdy nebo zda bude zpráva skutečně přečtena, což kompletně závisí na chování příjemce. [2]

Obrázek 13: Operace fronty zpráv [27]

Tyto systémy zpravidla zahrnují několik klíčových prvků, které společně zajišťují správné doručení zpráv, jejich správu a zpracování. Mezi hlavní komponenty obdobně jako u event-based architektury patří [2], [26]:

- **Broker** – broker zpráv je centrálním uzlem v pozici prostředníka mezi producenty a konzumenty. Přijímá zprávy od producentů, spravuje je ve frontách a následně je distribuuje příslušným konzumentům. Dále může také poskytovat další funkce jako je zajištění trvalosti zpráv, zajištění doručení, případně transformace zpráv a routing založený na obsahu nebo prioritě zpráv.
- **Producent** – aplikace nebo služby, které generují zprávy a odesílají je do systému front zpráv. Mohou to být webové servery, databázové systémy nebo jakékoliv jiné aplikace, které potřebují komunikovat se zbytkem systému.
- **Konzument** – aplikace nebo služby, které přijímají zprávy z fronty. Konzumenti zpracovávají zprávy podle svých specifických úkolů, které mohou zahrnovat vše od jednoduchých úprav dat, po komplexní transakční zpracování nebo dávkové operace.
- **Fronty zpráv** – datové struktury, které slouží jako úložiště pro zprávy čekající na zpracování. Zprávy v frontě jsou obvykle organizovány podle principu FIFO (First-In, First-Out), což znamená, že první zpráva, která byla vložena do fronty, bude také první zpracována.

- **Správce front** – komponenta řídící vytváření, odstraňování a správu front zpráv. Monitoruje stav front a zajišťuje, že zprávy jsou uchovávány bezpečně a efektivně.

1.3.4 Stream-Oriented

Streamově orientovaná komunikace v distribuovaných systémech umožňuje spojitý a real-time tok dat mezi producenty a konzumenty, což je zásadní pro aplikace vyžadující okamžité zpracování dat. Tento způsob komunikace se vyznačuje nepřetržitým a inkrementálním zpracováním dat, které přicházejí ve velkém objemu a umožňuje real-time analýzy a monitorování systémů s nízkou latencí.

V praxi se streamově orientovaná komunikace uplatňuje například ve finančních platformách pro rychlé obchodování, v IoT prostředí pro správu senzorových dat, nebo v aplikacích pro zpracování real-time dat z různých zdrojů. Klíčové komponenty systému zahrnují producenty dat, jako jsou senzory nebo uživatelská rozhraní, konzumenty dat, kterými mohou být analytické nástroje nebo databáze, a streamovací enginy, které řídí tok dat mezi producenty a konzumenty. [1], [12]

1.3.5 Multicast

Multicastová komunikace ve distribuovaných systémech umožňuje odesílání zpráv z jednoho zdroje (odesílatele) skupině procesů, čímž efektivně šíří data mezi více příjemci. Tento způsob komunikace se hodí pro aplikace, které vyžadují simultánní distribuci obsahu, jako jsou živé přenosy nebo skupinové konference.

Základem multicastové komunikace je využití speciální multicastové adresy, která funguje jako filtr pro procesy ve skupině: procesy, které naslouchají na této adrese, jsou považovány za členy skupiny a přijímají zprávy, zatímco ostatní procesy zprávy ignorují. Praktická realizace multicastu zahrnuje nastavení komunikačních cest, což může být administrativně náročné a často vyžaduje manuální zásahy. [2]

2 AMQP

Protokol AMQP (Advanced Message Queuing Protocol) je standardní komunikační protokol navržený pro zajištění interoperability mezi různými zprávovými systémy. Vznikl jako odpověď na potřebu unifikace a standardizace řešení komunikace pomocí front zpráv, které byly často proprietární a nekompatibilní. AMQP definuje robustní a flexibilní rámec pro zasílání zpráv, který podporuje jak synchronní, tak asynchronní komunikaci mezi distribuovanými systémy. [2]

Komunikace v AMQP probíhá obdobně jako bylo popsáno v kapitole o komunikaci pomocí modelu front zpráv. AMQP modeluje komunikaci pomocí základních komponent, které jsou popsány v kapitole 2.1. Významným příkladem implementace AMQP je RabbitMQ, který podporuje různé verze tohoto protokolu, včetně verze 0-9-1 která byla použita v této diplomové práci. [2]

2.1 Komponenty protokolu AMQP

Tato kapitola byla zpracována dle zdrojů [2], [28] a [29]. Jak již bylo zmíněno výše, komponenty protokolu AMQP jsou obdobné komunikačnímu modelu front zpráv. Zároveň však protokol představuje i nové součásti jako například exchanges, které jsou stěžejní pro průběh komunikace. Jelikož je to síťový protokol, tak všechny jeho komponenty mohou být rozmístěny po síti na různých zařízeních.

- **Brokery** – serverové aplikace, které řídí komunikaci v systému. Přijímají zprávy od producentů a směrují je konzumentům. Broker je zodpovědný za celkové řízení a ukládání zpráv. Zahrnuje správu spojení, front, transakcí, atp.
- **Exchanges** – Exchange je součást brokeru, která je zodpovědna za směrování zpráv do jedné nebo více front, na základě specifických pravidel a kriterií (například směrovací klíče nebo vazby). Rozlišujeme několik typů exchanges:
  - *Direct exchange* = přímé routování, které směruje zprávy do front na základě shody směrovacího klíče.
  - *Fanout exchange* = rozesílá zprávy do všech připojených front bez ohledu na směrovací klíč.

  - *Topic exchange* = používá vzory směrovacích klíčů pro směrování zpráv do jedné nebo více front.
  - *Headers exchange* = routuje zprávy na základě shody hlaviček zprávy s hodnotami specifikovanými při vytváření vazby.
- **Fronty** – datové struktury shromažďující zprávy, které čekají na zpracování konzumenty. Každá fronta musí být zvlášť deklarována a lze jí nastavit specifické vlastnosti jako je název, trvanlivost (zda přežije restart brokeru), exkluzivita, auto-delete a další.
- **Konzumenti** – aplikace, které se pomocí operace subscribe registrují k odebírání zpráv z jedné nebo více front. Po přijetí zprávy mohou odeslat acknowledgement k potvrzení úspěchu pro broker a producentskou aplikaci. Je také možné odmítnout zprávu pomocí odeslání negative acknowledgement – taková zpráva je dle nastavení buď zahozena nebo zařazena zpět do fronty na nejbližší možné místo tomu původnímu.
- **Producenti** – pomocí operace publish odesílají data do front. Dále popsáno v kapitole 1.3.3 v části *Fronty zpráv*.
- **Vazby** – neboli bindings jsou pravidla, která používají exchanges k určení, do kterých front mají být zprávy směrovány.

2.2 Průběh komunikace

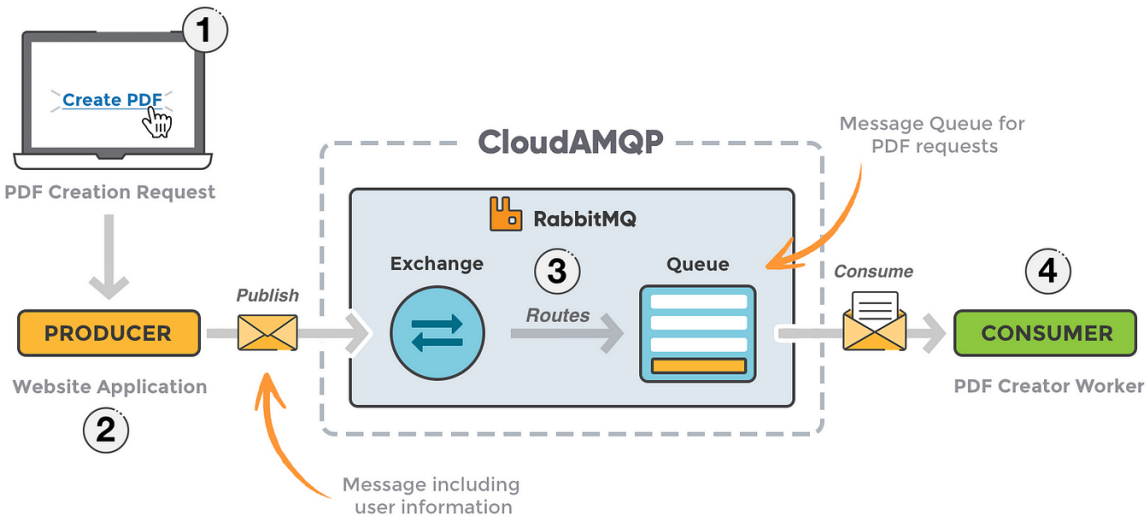
Tato kapitola byla zpracována dle RabbitMQ dokumentace o implementaci protokolu AMQP. [28]

AMQP připojení v aplikacích jsou obvykle dlouhodobá, využívající protokol TCP s autentizací využívající chráněnou vrstvu TLS.

Některé aplikace mohou potřebovat více než jedno připojení k brokeru a jelikož není žádoucí udržovat mnoho TCP připojení současně kvůli nadměrné spotřebě prostředků, řešením jsou AMQP kanály sdílející jedno TCP připojení. Všechny probíhající operace na klientovi se provádějí přes kanál. Kanály mají vlastní ID a veškerá komunikace je na všech kanálech separátní.

Když aplikace potřebuje zaslat zprávu (publish), spojí se s brokerem a použije exchange k definování, kam má být zpráva poslána. Exchanges rozhodují, do kterých front zprávy patří, na základě pravidel vazeb a typu exchange. Konzumenti se přihlásí k odběru zpráv

z front (subscribe), což může být provedeno buď předem (push model) nebo na vyžádání (pull model).



Obrázek 14: Cloud AMQP [27]

AMQP podporuje potvrzování doručení zpráv, kdy konzument potvrdí přijetí zprávy, což brokerovi signalizuje, že zpráva může být z fronty odstraněna. V situacích, kdy zpráva nemůže být doručena nebo zpracována, je zpráva buď vrácena producentovi, zahozena nebo umístěna do fronty mrtvých zpráv (dead letter queue).

Pokud již aplikace nepotřebuje připojení k serveru, měla by řádně ukončit AMQP spojení místo náhlého ukončení základního připojení TCP.

3 DOCKER

Docker je open-source platforma, která s využitím kontejnerové virtualizace umožňuje vývojářům vytvářet, spouštět a testovat aplikace či služby v izolovaném prostředí. Tato izolace zjednodušuje správu a zabezpečuje konzistenci prostředí napříč vývojovým cyklem aplikace. Docker kontejnery jsou malé a obsahují všechny potřebné závislosti aplikace jako jsou knihovny a systémové nástroje, díky čemuž jsou zaručeně spustitelné v jakémkoliv prostředí. Technologie Docker od svého uvedení v roce 2013 nabízí vývojářům efektivní nástroje pro rychlé nasazení a škálování aplikací, ať už lokálně, v cloudovém prostředí, nebo ve smíšených prostředích. [30], [33], [31]

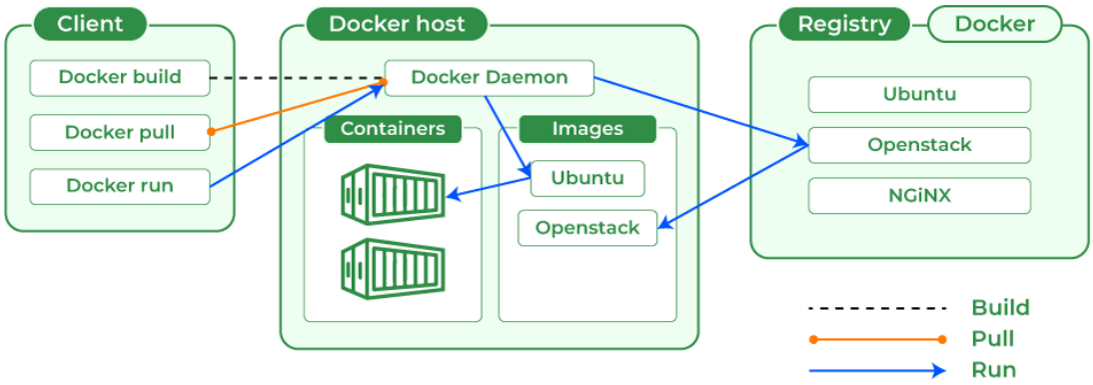
Kontejnerová virtualizace

Kontejnerová virtualizace, jak ji používá Docker, je forma virtualizace na úrovni jádra operačního systému, která umožňuje spouštění služeb v izolovaných kontejnerech sdílejících jádro hostitelského systému. Tato technika šetří systémové zdroje oproti tradiční virtualizaci, kde je pro každou službu nutné virtualizovat celý operační systém.

Kontejnerová virtualizace zahrnuje izolaci procesů, souborových systémů a síťového provozu, což zajišťuje, že výkon služeb je srovnatelný s jejich během v nativním prostředí. Hlavní nevýhodou je nižší úroveň bezpečnosti, protože selhání jádra ovlivňuje všechny běžící služby. [34], [32]

3.1 Architektura a komponenty

Docker používá klient-server architekturu. Docker Client (klientská část) slouží jako primární uživatelské rozhraní, umožňující vývojářům komunikovat s daemonem prostřednictvím příkazů nebo REST API. Docker Daemon (serverová část) spravuje stavy kontejnerů, obrazy a sítě, a také zajišťuje sestavování, běh a distribuci docker kontejnerů. Klient a démon mohou běžet na stejném systému, nebo je také možné připojit klienta Docker ke vzdálenému démonu. [30], [31]



Obrázek 15: Docker architektura [35]

- **Images (obrazy)** – Šablony s instrukcemi pro vytváření kontejnerů, skládající se z vrstev definovaných v Dockerfile, což umožňuje rychlé sestavení a úpravy. [30]
- **Kontejnery** – Spustitelné instance obrazů, které lze spravovat pomocí Docker CLI nebo API. Jsou izolované a umožňují připojení k sítím a úložištím. [30]
- **Volumes** – Trvalé a spravovatelné úložiště dat mimo životní cyklus kontejneru. Volumes umožňují ukládat a spravovat data generovaná kontejnery a zároveň poskytují možnost sdílet data mezi kontejnery nebo mezi kontejnery a hostitelským systémem. [30]
- **Sítě** – Síťový subsystém dockeru umožňuje propojovat kontejnery mezi sebou nebo i s externími aplikacemi. Subsystém funguje na principu ovladačů pomocí kterých je možné nakonfigurovat komunikaci mezi kontejnery jak na jednom hostitelském zařízení tak i napříč různými hostiteli. Konfigurace sítě se provádí pomocí příkazů docker network, což umožňuje uživatelům přizpůsobit síťové nastavení dle potřeb. [36]
- **Docker Compose** – Docker Compose je nástroj pro definici a správu vícekontejnerových Docker aplikací pomocí YAML souboru, který obsahuje veškeré potřebné konfigurace pro nasazení kontejnerů. Integrovaný s Docker Swarm, Docker Compose umožňuje snadné sestavení a nasazení kontejnerů na jednom hostiteli. [35]
- **Docker Desktop** – Aplikace umožňující snadnou instalaci a správu Dockeru na operačních systémech Mac, Windows nebo Linux. Obsahuje Docker daemon (dockerd), Docker klient (docker), Docker Compose, Docker Content Trust, Kubernetes a Credential Helper, což usnadňuje vývoj a sdílení kontejnerizovaných aplikací a mikroslužeb. [30]

- **Docker Registry** – Docker registry, jako je Docker Hub, je cloudové úložiště, kde uživatelé mohou nahrávat a stahovat Docker obrazy. Tyto registry mohou být veřejné nebo soukromé a slouží k snadnému sdílení a znovupoužití obrazů. Docker automaticky vyhledává obrazy na Docker Hub, a umožňuje uživatelům používat příkazy jako docker pull nebo docker push pro manipulaci s obrazy v rámci nakonfigurovaného registry. [30]

4 XML FORMÁT

XML, neboli Extensible Markup Language, je značkovací jazyk, který umožňuje definovat a ukládat data ve strukturované formě. Data jsou díky tomu snadno sdílitelná mezi různými počítačovými systémy jako jsou například webové stránky, databáze a aplikace třetích stran. Tento formát je čitelný jak pro počítačové systémy, tak i pro lidi.

Na rozdíl od HTML, které formátuje zobrazení webové stránky, XML se zaměřuje na organizaci informací, nikoli na jejich vzhled. To umožňuje XML definovat vlastní značky (tagy), které popisují význam nebo použití dat. Díky samopopisnosti tohoto jazyka v sobě každý XML dokument nese definici své vlastní struktury, což usnadňuje interpretaci dat nezávisle na aplikaci, která je používá. [37], [38]

4.1 Struktura XML

Formát XML organizuje své prvky do hierarchické stromové struktury. Každý XML dokument začíná deklarací, která specifikuje metadata dokumentu jako je verze XML a použité kódování znaků. Tato deklarace není povinná, ale je doporučena pro zajištění kompatibility a správného zpracování dokumentu. Příklad standardní deklarace vypadá takto:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Kořenový element

Každý XML dokument musí mít právě jeden kořenový element, který obaluje všechny ostatní elementy. Kořenový element tedy tvoří základní strukturu dokumentu a všechny další elementy jsou jeho potomci.

Elementy a tagy

Elementy jsou základními stavebními bloky XML a jsou označovány značkami (tagy). Každý element má startovací tag, obsah a koncový tag. Název tagu je ohraničen těmito znaky <název> pokud se jedná o startovací tag. Koncový tag je ohraničen takto <název/>. Například:

```
<title>Everyday Italian</title>
```

Atributy

Elementy mohou mít také atributy, které poskytují dodatečné informace. Atributy jsou umístěny uvnitř startovacího tagu a mají formát název="hodnota". Atributů může mít jeden element i více. Příklad:

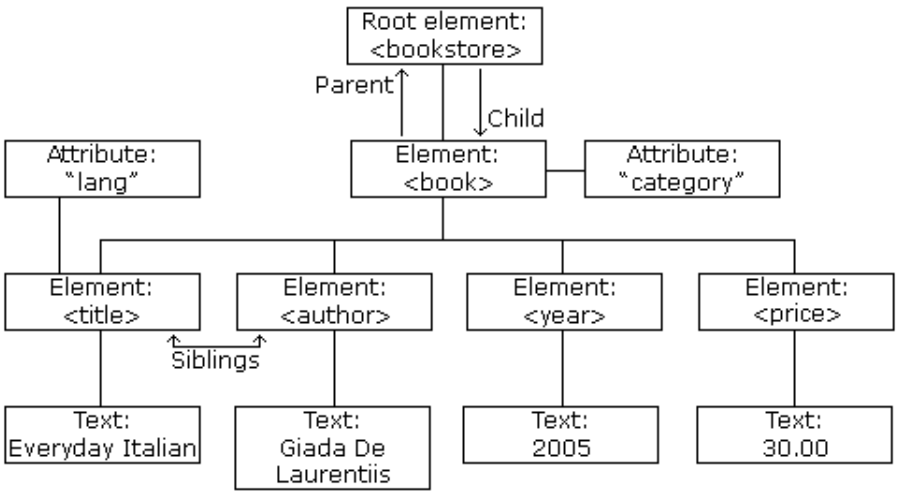
```
<book category="cooking">
```

Hierarchie

Tato kapitola je zpracována dle zdrojů [39] a [40]. Jak zde již bylo zmíněno, XML dokumenty jsou strukturovány jako stromy. Kořenový element slouží jako kořen stromu, do kterého jsou vloženy všechny ostatní elementy jako jeho větve. Tato hierarchie usnadňuje navigaci v dokumentu a manipulaci s daty.

Každý element může mít několik potomků (child elements), kteří jsou zanořeni v rámci jeho tagů. Dále elementy, které sdílejí stejného rodiče, jsou považovány za sourozence. V XML dokumentech je běžné, že sourozenecké elementy reprezentují opakované položky nebo položky na stejné úrovni.

Pro demonstraci hierarchie zápisu XML je zde uveden příklad XML dokumentu z webové stránky w3schools obsahující tutoriál k jazyku XML. Na obrázku 16 je možné vidět vizualizaci hierarchie elementů v XML dokumentu, jehož příklad je vložen pod obrázkem. Obrázek znázorňuje, které elementy dokumentu mají mezi sebou vztahy rodič, potomek, sourozenec. Dále jsou znázorněny i atributy a hodnoty elementů.



Obrázek 16: Hierarchie XML [40]

V tomto případě je kořenovým prvkem *bookstore* a pod něj spadají všechny ostatní prvky. Jak je z obrázku 16 a následujícího XML dokumentu zřejmé, všechny prvky typu *book* jsou v roli potomků kořenového prvku *bookstore*. Dále je možné vyčíst, že každý prvek typu *book* má atribut *category* a obsahuje potomky *title*, *author*, *year* a *price*.

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

XML Namespaces

XML Namespaces, pomáhají předcházet konfliktům v názvech elementů v XML dokumentech tím, že umožňují definovat jmenné prostory pomocí URI a prefixů ve formě atributů "xmlns". Toto řešení zajišťuje jasné rozlišení elementů s totožnými názvy, ale odlišnými významy a kontexty, což usnadňuje integraci a zpracování dat z různých zdrojů. Příklad deklarace namespace je v následujícím bloku kódu. [41]

```
<root xmlns:h="http://www.w3.org/TR/html4/"
      xmlns:f="https://www.w3schools.com/furniture">
  ...
</root>
```

XLink

XLink je technologie používaná k vytváření hypertextových odkazů v XML dokumentech. Umožňuje, aby jakýkoliv element v XML dokumentu fungoval jako odkaz, a to i mimo samotné svázané soubory. XLink odkazy jsou definovány přidáním atributů z namespace XLink, který je identifikován URI "http://www.w3.org/1999/xlink". Aby se aktivovaly funkce XLink, musí být toto namespace deklarováno v dokumentu. [42]

Ukázka použití odkazování pomocí XLink z tutoriálu w3schools [42]:

```
<?xml version="1.0" encoding="UTF-8"?>
<homepages xmlns:xlink="http://www.w3.org/1999/xlink">
<homepage xlink:type="simple"
            xlink:href="https://www.w3schools.com">W3Schools</homepage>
</homepages>
```

XPath

XPath je jazyk určený k navigaci a adresaci specifických elementů z XML dokumentů. Umožňuje definovat cesty k jednotlivým uzlům nebo skupinám uzlů v XML struktuře pomocí definovaných výrazů. Tento nástroj je podporován v mnoha programovacích jazycích a je doporučením konsorcia W3C. [43]

Tabulka 1: Přehled hlavních XPath výrazů pro navigaci v XML dokumentech. [44]

Výraz	Popis
`nodename`	Vybere všechny uzly s názvem *"nodename"*.
`/`	Vybere od kořenového uzlu, definuje absolutní cestu.
`//`	Vybere uzly kdekoliv v dokumentu od aktuálního uzlu.
`.`	Vybere aktuální uzel.
`..`	Vybere rodiče aktuálního uzlu.
`@`	Vybere atributy.

4.2 XSD schéma

XSD, známé také jako XML Schema Definition, je nástroj pro popis struktury a ověřování formátu XML dokumentů. Na rozdíl od DTD (Document Type Definitions) je XSD psáno v XML a podporuje širší škálu datových typů a jazykových konstrukcí. Hlavním účelem používání XML schématu je zajištění, že XML dokumenty splňují definované struktury a pravidla. XSD poskytuje mechanismus pro [45]:

- Definici prvků a atributů, které se mohou objevit v dokumentu.
- Počet a pořadí potomků daného prvku.
- Specifikaci datových typů pro prvky a atributy, což zahrnuje omezení na textový obsah.
- Vytvoření výchozích a pevných hodnot pro prvky a atributy.

Struktura XSD

Jak již bylo zmíněno, XSD má stejnou syntaxi jako XML. Dále schéma podporuje dědičnost, díky které umožňuje znovupoužitelnost již definovaných částí na více místech. [46]

Základní prvky XSD zahrnují elementy a typy jako:

- <xs:element>,
- <xs:complexType>,
- <xs:simpleType>.

Mezi běžné datové typy podporované v XSD patří:

- xs:string,
- xs:decimal,
- xs:integer,
- xs:boolean,
- xs:date,
- xs:time.

Složité typy (<xs:complexType>) mohou obsahovat jiné elementy a atributy, zatímco jednoduché typy (<xs:simpleType>) definují omezení pro textový obsah a atributy. Například, v následujícím kódu je definován element note, který obsahuje čtyři další elementy (to, from, heading, body), každý s typem xs:string [47]:

```
<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

5 NÁVRH SYSTÉMU

Tato kapitola popisuje požadované vlastnosti praktické části diplomové práce. Dále pojednává o návrhu řešení distribuovaného systému splňujícího požadavky ze zadání a přibližuje problematiku geodetických kontrol v kapitole 5.3. Implementace geodetických kontrol není součástí této práce. Problematika kontrol je zde jen obecně popsána pro vysvětlení kontextu, jelikož jsou zde kontroly často zmiňovány díky tomu, že systém byl implementačně přizpůsoben právě pro provozování těchto kontrol.

5.1 Požadované vlastnosti

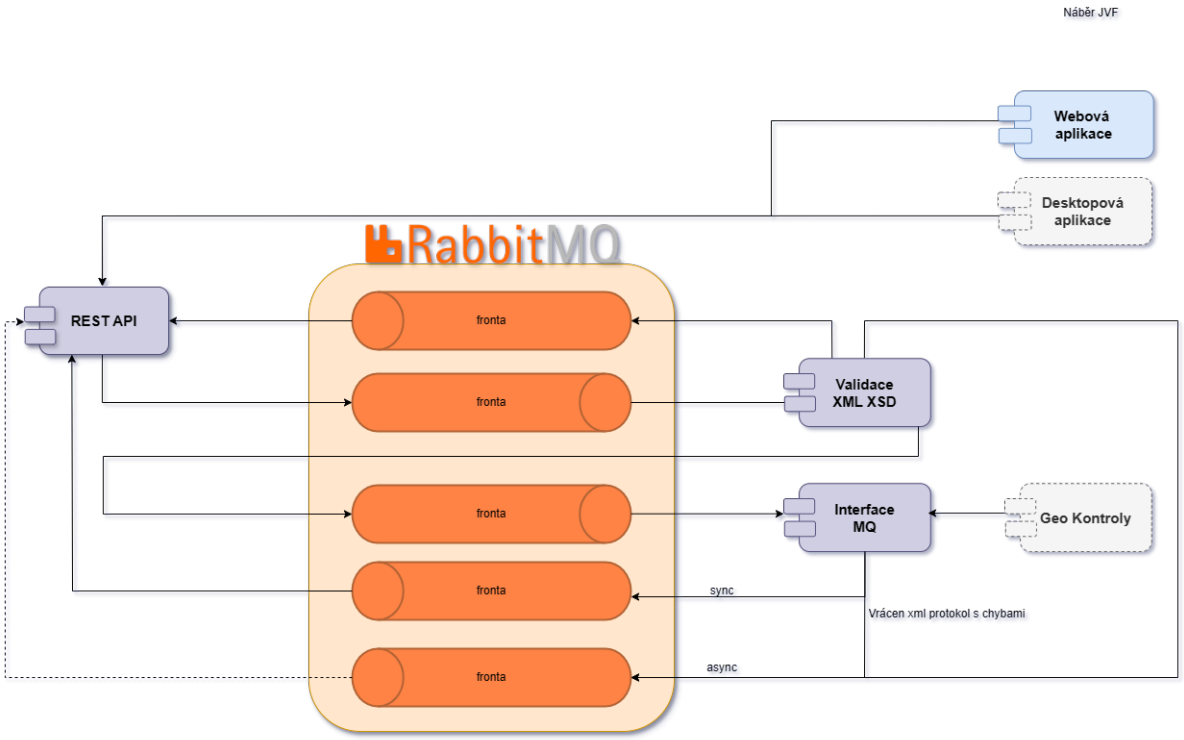
Cílem této diplomové práce je vytvoření uceleného systému pro kontrolu geodetických dat v distribuovaném prostředí. Požadavky na výsledný systém zahrnují tyto vlastnosti:

- splnění vlastností/principů distribuovaného systému,
- postaveno na platformě .NET,
- komunikace pomocí protokolu AMQP,
- kontejnerizace pomocí Docker kontejnerů,
- poskytuje uživatelské rozhraní pro zadání kontroly a zobrazení výsledků.

5.2 Návrh řešení

Výsledný distribuovaný systém byl navržen jako hybridní architektura využívající vlastnosti mikroservisní a zprávově-orientované architektury. Hlavním prvkem distribuovaného systému je RabbitMQ poskytující fronty zpráv pro komunikaci pomocí AMQP protokolu a broker řídící komunikaci v systému. Pro dodržení podmínky decentralizace je možné RabbitMQ broker nakonfigurovat do clustering režimu, což umožňuje distribuovat broker mezi několik zařízení a odstranit tak problém jediného bodu selhání systému. Tato komponenta tedy zahrnuje zprávově orientovanou část architektury.

Jednotlivé komponenty zastávající specifické funkce v systému jsou v architektuře implementovány jako mikroslužby, které mezi sebou komunikují pomocí zasílání správ do definovaných front. Konkrétně byly pro implementaci samostatných mikroslužeb vytvořeny tyto aplikace: aplikace poskytující Rest API, aplikace pro validaci XML dat proti XSD schématu a aplikace v roli rozhraní volajícího geodetické kontroly.



Obrázek 17: Schéma navrženého systému. Zdroj vlastní.

Jak je možné vidět z obrázku 17, vstupním bodem do systému může být jak webová aplikace, tak desktopová aplikace. Uživatel nahraje pomocí formuláře data a pošle je ke kontrole. Dle zvoleného typu kontroly je volán specifický endpoint API, ze kterého jsou data následně odeslána do konkrétní RabbitMQ fronty. Z té si data vyzvedne mikroslužba pro validaci dat. Pokud jsou data nevalidní, dále se nepokračuje a systém vrátí uživateli přes API informaci o chybě ve struktuře XML dat. Pokud jsou data validní, tak jsou odeslána do fronty, ze které data odebírá aplikace pro volání a provedení geodetických kontrol. Výsledky jsou uživateli vráceny podle typu volání buď synchronně zpět do aplikace nebo jsou uloženy do fronty výsledků a poskytnuty uživateli asynchronně na vyžádání.

5.3 Geodetická datová kontrola

Jak zde již bylo zmíněno, algoritmy geodetických datových kontrol nejsou součástí této diplomové práce. Avšak jelikož byl distribuovaný systém implementován na míru provozování těchto kontrol, je dobré zmínit o co se jedná. Dále budou v této kapitole také popsány datové formáty, které jsou používány pro vstupní a výstupní data implementovaného distribuovaného systému.

Geodetické datové kontroly zajišťují správnost a přesnost geodetických dat před jejich finálním zpracováním a začleněním do geodetických informačních systémů. Proces kontroly geodetických dat obvykle zahrnuje několik úrovní ověřování od základních kontrol až po složité topologické kontroly. Kontrolují se jak nastavené atributy prvků, tak i jejich geometrie.

Nejprve se provádějí základní kontroly, které zahrnují kontrolu validity XML dat vůči komplexnímu XSD schématu, dále probíhá ověření správného rozvrstvení dat, typů geodetických prvků, souladu seznamu souřadnic s výkresem a kontrolu délky úseček. Tyto kontroly jsou zaměřeny na formální aspekty dat a jsou prvotním krokem v procesu validace. [48]

Kromě těchto kontrol se také provádějí specializované kontroly, jako je kontrola atributů, která zajišťuje správné naplnění atributů objektů, kontrola geometrií, která ověřuje použití pouze povolených typů geometrií, a kontrola umístění dat, která zjišťuje správné geografické umístění dat v rámci příslušného regionu. [48], [49]

Na druhé úrovni se provádějí složitější topologické kontroly, které kontrolují samotné geometrie prvků. Tyto kontroly zahrnují detekci křížení a překrývání linií, duplicitních bodů a buněk, blízkosti bodů a buněk, stejně jako kontrolu volných konců linií a lomových bodů, kontroly ploch a mnoho dalších. Tyto kontroly jsou klíčové pro zajištění, že geodetická data jsou topologicky konzistentní, mohou být správně interpretována a využita v dalších fázích zpracování. [50]

5.3.1 Použité datové formáty

Pro vstupní i výstupní data v implementovaném datovém systému byla použita XML data ve standardizovaném formátu, který je využíván v rámci projektu DTM ČR – Digitální Technické Mapy ČR.

JVF DTM

Jednotný výměnný formát JVF DTM je standardizovaný datový formát určený pro výměnu geodetických dat mezi veřejnou správou, geodety a dalšími uživateli. Tento formát umožňuje efektivní sdílení a aktualizaci dat digitálních technických map (DTM) a je zásadní pro správu, údržbu a rozvoj zeměpisných informačních systémů.

Formát JVF DTM vychází z vyhlášky o digitální technické mapě a je modelován tak, aby podporoval strukturovanou organizaci geodetických dat do kategorií, skupin a typů objektů. Všechna data jsou uložena ve formátu XML, což umožňuje jejich přehledné uspořádání a snadnou validaci pomocí XSD souborů, které definují strukturu a pravidla pro zápis dat.

Hlavní součástí souboru JVF DTM jsou data o objektech, která zahrnují základní informace o typu objektu, jeho specifických atributech a jeho geometrii, která splňuje schéma GML (Geography Markup Language). Kromě toho formát obsahuje servisní informace pro podporu systému DTM a extenze, které umožňují rozšíření standardního obsahu dle potřeb uživatelů. [51], [52]

V následujícím bloku kódu je ukázka podoby souboru JVF DTM jako vstupních dat do distribuovaného systému poskytujícího geodetické kontroly.

```
<?xml version="1.0" encoding="utf-8"?>
<JVFDTM xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:gml="http://www.opengis.net/gml/3.2" xmlns="objtyp">
  <DataJVFDTM>
    <VerzeJVFDTM xmlns="cmn">1.4.2.3</VerzeJVFDTM>
    <DatumZapisu xmlns="cmn">2024-01-11T11:14:45.333+01:00</DatumZapisu>
    <TypZapisu xmlns="cmn">kompletní zápis</TypZapisu>
    <Data>
      <PodrobnyBodZPS>
        <ObjektovyTypNazev code_base="0100000218" code_suffix="01"
        xmlns="pobzps">podrobný bod ZPS</ObjektovyTypNazev>
        <KategorieObjektu xmlns="pobzps">
            Geodetické prvky
        </KategorieObjektu>
        <SkupinaObjektu xmlns="pobzps">Podrobný bod</SkupinaObjektu>
        <ObsahovaCast xmlns="pobzps">ZPS</ObsahovaCast>
        <ZaznamyObjektu xmlns="pobzps">
          <ZaznamObjektu>
            <ZapisObjektu xmlns="cmn">i</ZapisObjektu>
            <AtributyObjektu>
              <SpolecneAtributyVsechObjektu xmlns="atr">
                <ID />
                <IDZmeny>CZ053-532806</IDZmeny>
                <PopisObjektu />
                <IDEditora />
                <DatumVkladu>2024-01-11T12:11:55.346+01:00</DatumVkladu>
                <VkladOsoba />
                <DatumZmeny>2024-01-11T12:11:55.346+01:00</DatumZmeny>
                <ZmenaOsoba />
```

```
            <DatumPlatnosti>2024-01-11T11:11:55.346</DatumPlatnosti>
          </SpolecneAtributyVsechObjektu>
          <SpolecneAtributyObjektuZPS xmlns="atr">
            <UrovenUmisteniObjektuZPS>0</UrovenUmisteniObjektuZPS>
          </SpolecneAtributyObjektuZPS>
          <TridaPresnostiPoloha xmlns="atr">3</TridaPresnostiPoloha>
          <TridaPresnostiVyska xmlns="atr">3</TridaPresnostiVyska>
          <ZpusobPorizeniZPS xmlns="atr">1</ZpusobPorizeniZPS>
          <CisloBodu xmlns="atr" />
        </AtributyObjektu>
        <GeometrieObjektu>
          <gml:pointProperty>
            <gml:Point gml:id="ID1" srsName="EPSG:5514"
                srsDimension="3">
              <gml:pos>-645410.73 -1061729.83 226.41</gml:pos>
            </gml:Point>
          </gml:pointProperty>
        </GeometrieObjektu>
      </ZaznamObjektu>
    ...
```

Na ukázce je možné vidět podobu definice souboru JVF DTM. V datové části souboru je jako příklad uveden zápis pro objekty typu PodrobnyBodZPS, atributy pro tento typ a záznam jednoho konkrétního prvku patřícího do této kategorie s vlastními atributy a geometrií typu gml:Point. Každý prvek disponuje i vlastním, v rámci souboru unikátním, atributem gml:ID.

Výstupní protokol

Výstupní protokol je definován ve formátu XML a slouží k efektivnímu sdělení výsledků datových kontrol geodetických aktualizačních dokumentací (GAD). Tento dokument je v průběhu provádění kontrol a je strukturován podle pevně daného schématu, který umožňuje jednoznačnou identifikaci a klasifikaci zjištěných chyb.

V následujícím bloku kódu je uveden příklad podoby výstupního protokolu geodetických kontrol, a tedy i datového výstupu implementovaného distribuovaného systému.

```
<?xml version="1.0" encoding="utf-16"?>
<ProtokolChyb xmlns:gml="http://www.opengis.net/gml/3.2"
    xmlns="protokolChyb">
  <IDZmeny />
  <IDPozadavku>8d943777-d219-4f91-a81d-7e6648dadf2c</IDPozadavku>
  <Kontroly>
    <Kontrola>
```

```
      <IDBehuKontroly>
        5316bc10-d509-4c9b-aae0-e56939dd35c1
      </IDBehuKontroly>
      <Skupina>JVF</Skupina>
      <KodKontroly>1.4</KodKontroly>
      <DatumKontrolyZacatek>2024-04-26T15:06:11</DatumKontrolyZacatek>
      <DatumKontrolyKonec>2024-04-26T15:06:11</DatumKontrolyKonec>
      <StavKontrola>0</StavKontrola>
      <SeznamChyb />
    </Kontrola>
    <Kontrola>
      <IDBehuKontroly>
        7315d4b6-5ca5-432a-aedd-3248929157f5
      </IDBehuKontroly>
      <Skupina>TOP</Skupina>
      <KodKontroly>1.5</KodKontroly>
      <DatumKontrolyZacatek>2024-04-26T15:06:11</DatumKontrolyZacatek>
      <DatumKontrolyKonec>2024-04-26T15:06:11</DatumKontrolyKonec>
      <StavKontrola>2</StavKontrola>
      <SeznamChyb>
        <Chyba>
          <StavChyba>2</StavChyba>
          <PopisChyby>Výška mimo povolený limit.</PopisChyby>
          <IDObjekt />
          <GmlID>ID11</GmlID>
          <LokalizaceChyby>
            <Linie>
              <gml:curveProperty>
                <gml:LineString gml:id="ID11" srsName="EPSG:5514"
                    srsDimension="3">
                  <gml:posList>
                    -648118.79 -1075459.26 0 -648111.03 -1075459.26 0
                  </gml:posList>
                </gml:LineString>
              </gml:curveProperty>
            </Linie>
          </LokalizaceChyby>
        </Chyba>
      </SeznamChyb>
    </Kontrola>
  </Kontroly>
</ProtokolChyb>
```

Kořenový element tohoto XML dokumentu je element ProtokolChyb, ve kterém je popsána identifikace změnového požadavku a vnitřní identifikace procesu kontroly. Dále obsahuje seznam kontrol v elementu Kontroly, kde je každá kontrola detailně specifiko-

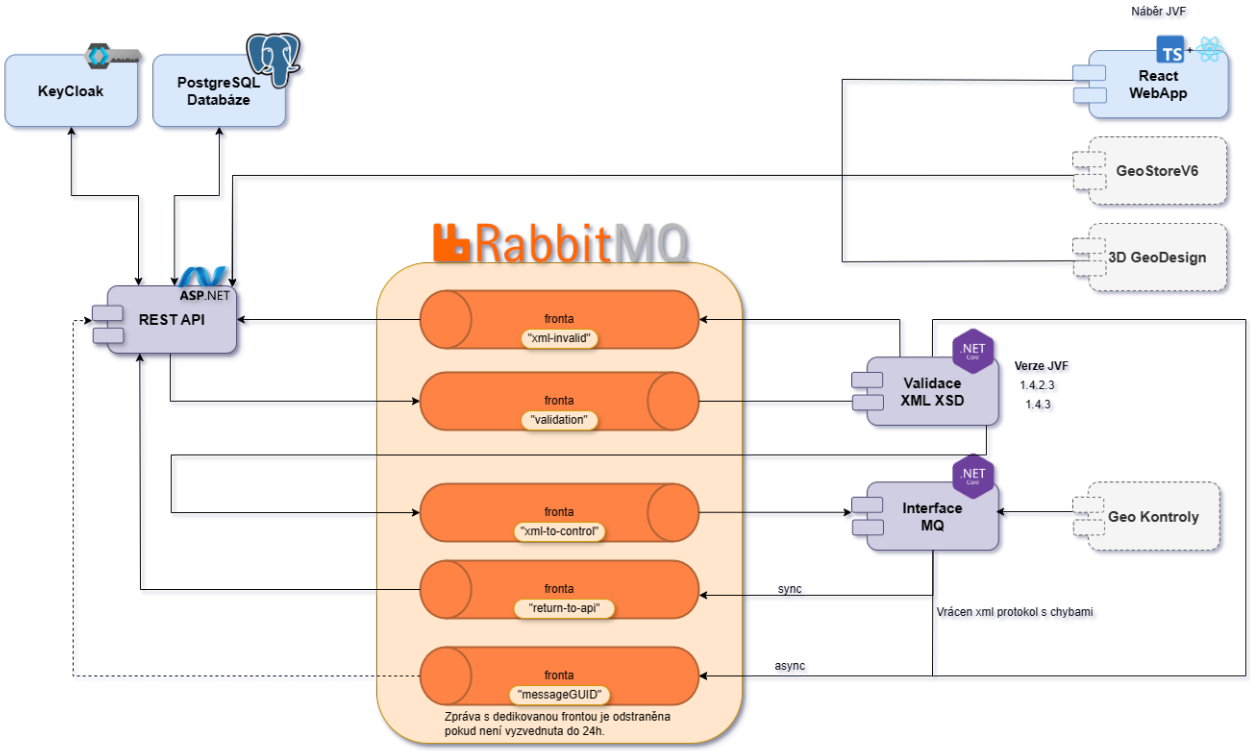
vána včetně svého jedinečného identifikátoru, skupiny, do které patří (např. atributové, topologické), a časového údaje o začátku a konci kontroly.

Každá kontrola může skončit stavem bez chyby, varováním nebo chybou, přičemž pro stavy varování a chyba je v dokumentu specifikován SeznamChyb. Tyto záznamy obsahují detailní popis identifikované chyby v datech, k jakému objektu chyba patří (pokud je uveden) včetně unikátního gml:ID objektu. Dále se uvádí také lokalizace chyby, která může být definována jako bod, linie nebo plocha, což zjednodušuje další analýzu a opravy dat.

Ve výstupním protokolu jsou tedy zaznamenány všechny provedené kontroly a jejich stav včetně kontrol bez chyb. Pokud kontrola skončila varováním nebo chybou, do protokolu jsou zaznamenány i chybové prvky, jejich geometrie a zprávy specifikujících danou chybu. [53]

6 IMPLEMENTACE SYSTÉMU

V této kapitole bude popsána implementace konkrétního řešení navrženého distribuovaného systému v předchozí kapitole, který kombinuje vlastnosti zprávově-orientované a mikroservisní architektury.



Obrázek 18: Schéma implementovaného systému. Zdroj vlastní.

Obrázek 18 modeluje podobu konkrétního řešení implementace distribuovaného systému. Jako vstup do distribuovaného systému je možné použít jak desktopovou, tak webovou aplikaci. Pro potřeby této diplomové práce byla vytvořena React Typescript webová aplikace poskytující uživateli přehled historie provedených kontrol a jednoduchý formulář pro vytvoření kontroly a odeslání dat. Správa uživatelů jako vytváření, úprava uživatelských účtů, přidělování oprávnění a ověřování identity uživatelů v systému je provozována pomocí Keycloaku, viz. kapitola 6.3. Data potřebná pro zobrazování informací ve webové aplikaci, uživatelská data a data kontrol jsou uchovávána v databázi PostgreSQL. Rest API poskytující endpointy pro volání jednotlivých skupin geodetických kontrol v synchronní i asynchronní variantě bylo implementováno jako ASP .NET REST API. Komunikaci v distribuovaném systému řídí technologie RabbitMQ, díky které jsou zprávy

mezi aplikacemi ukládány do specifických front. Dále je v systému implementována aplikace jako mikroslužba poskytující validaci XML vstupních dat pomocí XSD schématu, která je implementovaná na platformě .NET v jazyce C#. Stejně tak aplikace poskytující rozhraní geodetických kontrol, je implementována jako mikroslužba v jazyce C# na platformě .NET.

Všechny zmíněné komponenty distribuovaného systému jsou nasazeny v Docker kontejnerech. Zejména jednotlivé mikroslužby a RabbitMQ broker lze dle potřeby škálovat a spouštět na různých zařízeních. Tím je zajištěna distribuce systému, snadná škálovatelnost a odolnost vůči chybám nebo selhání konkrétního prvku systému.

6.1 RabbitMQ

Jednu ze stěžejních částí distribuovaného systému zastupuje technologie RabbitMQ, která byla použita pro implementaci komunikace v systému.

RabbitMQ je spolehlivý a pokročilý open-source broker pro zasílání a streamování zpráv. Podporuje různé komunikační protokoly jako například AMQP, MQTT, HTTP a další. Dále poskytuje také klientské knihovny pro širokou škálu programovacích jazyků. K jeho výhodám patří široká nabídka možností konfigurace, jakým způsobem budou předávány zprávy od producentů mezi konzumenty. Mezi tyto možnosti patří komplexní routování, streamování nebo například filtrování zpráv. Další výhodou RabbitMQ je jeho spolehlivost, která je zajištěna díky mechanismu potvrzování doručených zpráv, možnosti opakovaného doručování, či replikace zpráv napříč clusterem brokerů. [54], [55]

Docker konfigurace

Soubor Dockerfile obsahuje konfiguraci obrazu, který je spouštěn v kontejneru. Nový obraz je vytvořen podle obrazu rabbitmq ve verzi 3.13 včetně instalovaného pluginu pro management rozhraní, který je dostupný z veřejného repozitáře. Poté jsou nastaveny proměnné prostředí pro vytvoření výchozího uživatele, hesla a virtual host prostředí. Nakonec jsou nastaveny porty, které obraz využívá a příkaz, který je proveden při každém spuštění kontejneru.

Na následující ukázce je možné vidět podobu konfigurace Dockerfile souboru pro RabbitMQ:

```
FROM rabbitmq:3.13-management

MAINTAINER silvia.cmilanska@gmail.com

# Environment variables setting
ENV RABBITMQ_DEFAULT_USER=controlrabbit
ENV RABBITMQ_DEFAULT_PASS=7a53QO6ltMHv4EOf
ENV RABBITMQ_DEFAULT_VHOST=geocontrolsvc

# Expose the ports for AMQP protocol and management interface
EXPOSE 5672 15672

CMD ["rabbitmq-server"]
```

Poté zbývá nakonfigurovat docker-compose.yaml soubor, ve kterém je uvedeno nastavení jedné či více služeb, které mají být kontejnerizovány. Služba byla nazvána jako "rabbitmq", dále je uvedena cesta kde se nachází kořenový adresář pro daný kontejner a jeho Dockerfile, nastavení mapování portů jak pro AMQP protokol, tak pro management rozhraní. Pod položkou volumes se nachází nastavení cesty k lokálnímu adresáři, kam mají být ukládána data kontejneru. V docker-compose.yml souboru je tak uvedeno pro každou službu jméno kontejneru pomocí položky *container_name*, díky které jsou poté služby mezi sebou síťově propojeny. Název kontejneru totiž slouží jako hostname pro daný kontejner.

```
  rabbitmq:
    build: ./RabbitMQ
    hostname: my-rabbit
    container_name: rabbitmq
    ports:
      - "5672:5672"  # AMQP protocol
      - "15672:15672"  # Management interface
    environment:
      - RABBITMQ_DEFAULT_USER=controlrabbit
      - RABBITMQ_DEFAULT_PASS=7a53QO6ltMHv4EOf
      - RABBITMQ_DEFAULT_VHOST=geocontrolsvc
    volumes:
      - ./RabbitMQ/data:/var/lib/rabbitmq
    restart: unless-stopped
```

Výše byla popsána konfigurace jednoho RabbitMQ serveru. V případě tohoto systému bylo však zapotřebí vytvořit více instancí RabbitMQ, které jsou spolu propojeny v clusteru, aby bylo zamezeno vzniku jediného bodu selhání. Proto byl vytvořen následující

konfigurační soubor *rabbitmq.conf*, který je používán pro nastavení clusteru v kontejnerech RabbitMQ. Hodnota *classic_config* znamená, že je využíván klasický způsob konfigurace pro objevování a propojování uzlů v clusteru. Tento způsob se spoléhá na explicitní konfiguraci seznamu uzlů, které by měly tvořit cluster. Dále jsou specifikovány jednotlivé uzly clusteru, které běží na kontejnerech rabbit(2,3) a všechny mají uživatelské jméno rabbit. Poslední řádek definuje automatickou obnovu v případě problému s některým uzlem. V takovém případě mezi uzly dojde k automatickému synchronizování stavů tak, aby byl cluster znovu konzistentní.

```
cluster_formation.peer_discovery_backend = classic_config
cluster_formation.classic_config.nodes.1 = rabbit@rabbitmq
cluster_formation.classic_config.nodes.2 = rabbit@rabbitmq2
cluster_formation.classic_config.nodes.3 = rabbit@rabbitmq3
cluster_partition_handling = autoheal
```

Následující konfigurační soubor je Dockerfile, který vypadá u všech 3 uzlů RabbitMQ stejně. Na rozdíl od souboru Dockerfile, který byl popsán výše pro jediný uzel RabbitMQ, tento Dockerfile obsahuje navíc definici hodnoty pro RABBITMQ_ERLANG_COOKIE, která je důležitá pro tvorbu clusteru a definuje zda spolu specifické uzly mohou komunikovat.

```
FROM rabbitmq:3.13-management

MAINTAINER silvia.cmilanska@gmail.com

# Copy custom configuration file
COPY rabbitmq.conf /etc/rabbitmq/

# Set environment variables
ENV RABBITMQ_ERLANG_COOKIE=secretcookie
ENV RABBITMQ_DEFAULT_USER=controlrabbit
ENV RABBITMQ_DEFAULT_PASS=7a53QO6ltMHv4EOf
ENV RABBITMQ_DEFAULT_VHOST=geocontrolsvc

EXPOSE 5672 15672
```

Následující ukázka zobrazuje jak může vypadat konfigurace docker-compose.yaml souboru pro 3 provázané RabbitMQ uzly v clusteru. Jejich hostname korespondují s konfiguračním souborem rabbitmq.conf a každý uzel má specifikovaný vlastní port na zařízení.

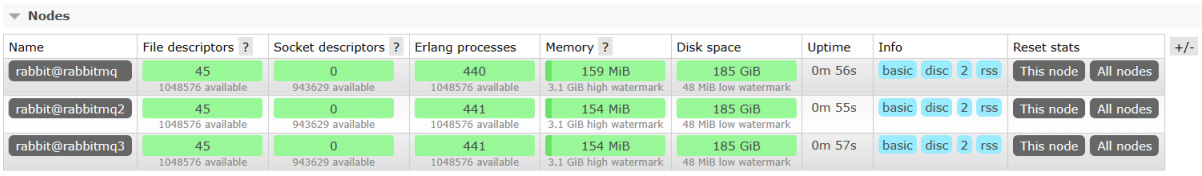
```yaml
rabbitmq:
  build: ./RabbitMQ
  hostname: rabbitmq
  container_name: rabbitmq
  ports:
    - "5672:5672"  # AMQP protocol
    - "15672:15672"  # Management interface
  volumes:
    - ./RabbitMQ/data:/var/lib/rabbitmq
  restart: unless-stopped

rabbitmq2:
  build:
    context: ./RabbitMQ2
  hostname: rabbitmq2
  container_name: rabbitmq2
  ports:
    - "5673:5672"
    - "15673:15672"
  volumes:
    - ./RabbitMQ2/data:/var/lib/rabbitmq
  restart: unless-stopped

rabbitmq3:
  build:
    context: ./RabbitMQ3
  hostname: rabbitmq3
  container_name: rabbitmq3
  ports:
    - "5674:5672"
    - "15674:15672"
  volumes:
    - ./RabbitMQ3/data:/var/lib/rabbitmq
  restart: unless-stopped
```

RabbitMQ management rozhraní

Následně je možné přihlásit se do management webového rozhraní. Jelikož bylo v mém případě konfigurováno na lokálním zařízení, tak bylo dostupné na následující adrese: http://localhost:15672/. Po zadání přihlašovacích údajů byl v záložce admin vytvořen nový uživatel, který bude používán pro připojení aplikací distribuovaného systému k RabbitMQ.

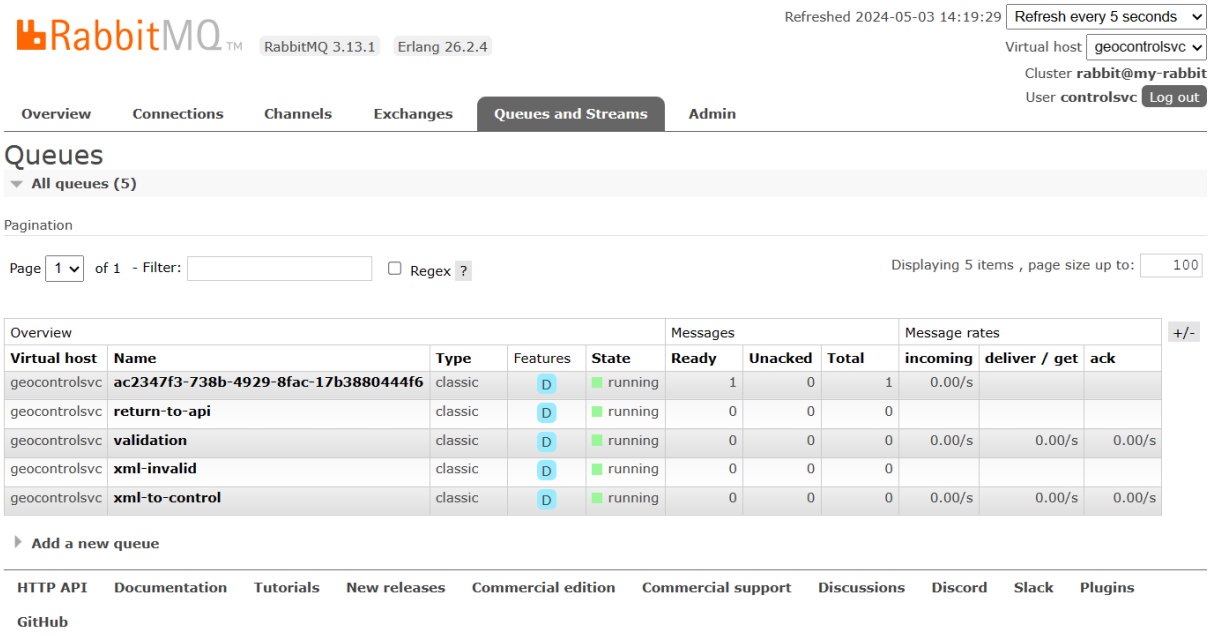
Obrázek 19: Přihlášení k RabbitMQ management rozhraní. Zdroj vlastní.

Další konfigurace již není potřebná, protože veškeré definice pro exchange a fronty jsou prováděny z klientských aplikací, což zajišťuje vytvoření potřebných objektů v RabbitMQ i v případě že by ve chvíli připojení klientské aplikace neexistovaly. Tím jsou eliminovány chyby ze strany konfigurace infrastruktury front.



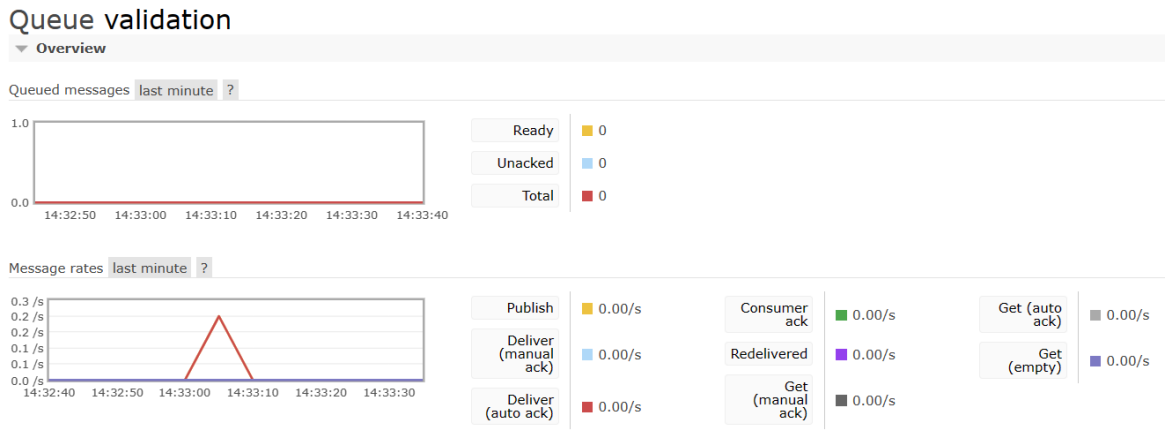
Obrázek 20: Aktivní uzly RabbitMQ clusteru. Zdroj vlastní.

Obrázek 20 zobrazuje vizualizaci připojených aktivních uzlů RabbitMQ clusteru v rámci dostupného management rozhraní. Administrátor zde může vyčíst užitečné informace o stavu a výkonu každého uzlu, které jsou důležité pro monitoring, ladění a optimalizaci.



Obrázek 21: RabbitMQ fronty. Zdroj vlastní.

Na obrázku 21 je demonstrována podoba management rozhraní s přehledem vytvořených front pro specifického virtuálního hostitele, kterým je v tomto případě *geocontrolsvc*. Na dashboardu je vidět forma routování pro každou frontu, což je zde konkrétně "direct", neboli přímé routování. Dále také stav fronty, počet zpráv ve frontě a rychlost přenosu zpráv.



Obrázek 22: Informační graf stavu fronty. Zdroj vlastní.

Z hlediska pohodlného řešení problémů ve frontách, zkoumání jejich obsahu a jejich správy je možné kliknout na požadovanou frontu, čímž se otevře stránka s přehledem informací o dané frontě. Stránka obsahuje přehledové informace včetně grafů počtu zpráv

ve frontě a rychlosti přijímání zpráv, ale také ovládací prvky pro zobrazení specifické zprávy z fronty, poslání zprávy, vyčištění fronty či její kompletní smazání a mnoho dalších.

6.2 Databáze

Pro implementaci databáze, která uchovává data potřebná pro zobrazování informací ve webové aplikaci, byla vybrána databáze PosgreSQL. PostgreSQL je výkonný otevřený objektově-relační databázový systém, který vyniká rozšířením standardního SQL a podporou komplexních datových struktur. Podporuje mnoho datových typů, včetně primitivních, strukturovaných a geometrických typů, a nabízí rozsáhlé možnosti pro zabezpečení a pokročilé indexování. [56]

Docker konfigurace

Databáze je nasazena v docker kontejneru, který je součástí konfiguračního souboru docker-compose.yaml obsahujícího konfiguraci několika souvisejících služeb. A to konkrétně tuto PostgreSQL databázi, službu Keycloak pro správu uživatelů a ASP .NET REST API, které využívá jak služby databáze tak Keycloaku.

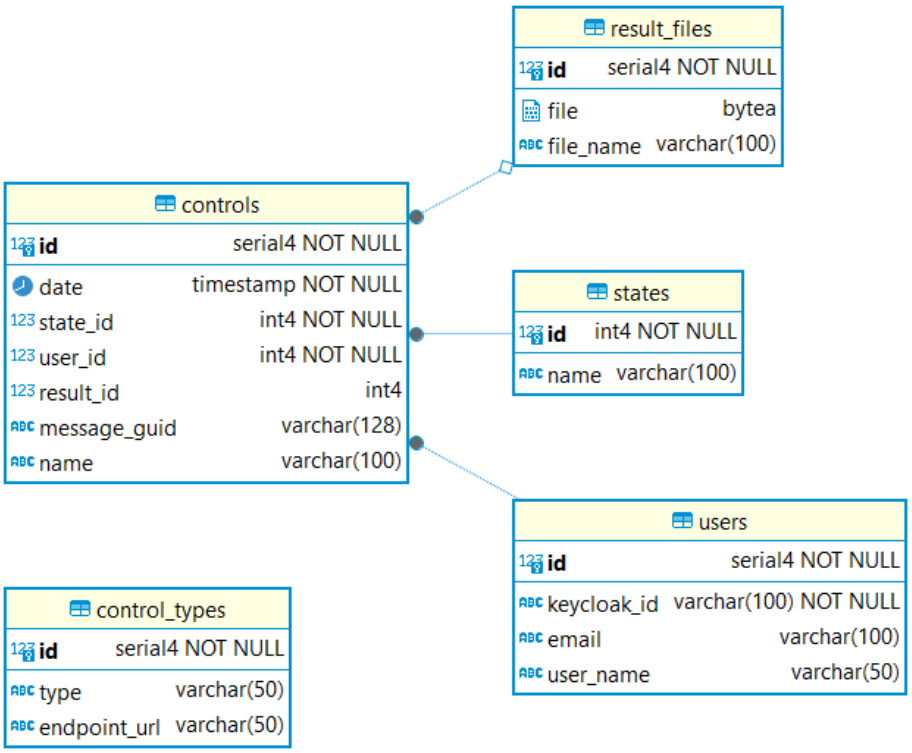
Databáze je postavena na obrazu aktuální verze PostgreSQL, dále obsahuje nastavení proměnných prostředí jako název databáze, a údaje uživatele. Dále mapuje vnitřní port kontejneru na defaultní port PostgreSQL, který bude otevřen Dockerem. Data kontejneru jsou díky volumes ukládána do adresáře ./postgres/data.

```
db:
  image: postgres
  container_name: db
  environment:
    POSTGRES_DB: geo_control_db
    POSTGRES_USER: geo_control_db
    POSTGRES_PASSWORD: geo_control_db
  ports:
    - "5432:5432"
  volumes:
    - ./postgres/data:/var/lib/postgresql/data
  restart: unless-stopped
```

Databázový model

Na obrázku 23 je možné vidět, že výsledný databázový model obsahuje 5 tabulek, z toho

dva číselníky. Prvním číselníkem je tabulka *control_types*, která obsahuje jednotlivé názvy spustitelných skupin kontrol a přiřazení API endpointu k dané skupině. Druhým číselníkem je tabulka *states*, která obsahuje názvy všech možných stavů výsledku geodetické kontroly.



Obrázek 23: Databázový model. Zdroj vlastní.

Hlavní tabulkou v databázi je tabulka *controls*, která obsahuje záznamy odeslaných a provedených kontrol. Na ni jsou napojeny tabulky *result_files*, *states* a *users*. Tabulka *result_files* obsahuje název a data výsledného protokolu, který je výstupem kontroly a tabulka *users* zaznamenává údaje o uživateli, které pomohou namapovat uživatelská data ke Keycloak uživatelské identitě.

6.3 Keycloak

Z důvodu integrace se stávajícími firemními aplikacemi, se kterými je tento distribuovaný systém propojen, byla pro vytváření, správu a ověřování uživatelů zvolena technologie Keycloak.

Jedná se o open-source nástroj pro správu identit a přístupů, vyvinutý společností Red Hat. Umožňuje snadné zabezpečení webových aplikací a RESTful webových služeb prostřednictvím jednotného přihlašování (Single Sign-On, SSO), což zjednodušuje proces autentizace uživatelů napříč různými aplikacemi. Keycloak podporuje mnoho standardních protokolů jako OpenID Connect, OAuth 2.0 a SAML 2.0, a díky tomu může efektivně integrovat uživatelské účty z různých zdrojů. [57], [58]

Docker konfigurace

Keycloak kontejner je založen na obraze Keycloaku verze 24.0.3. Kontejner je spouštěn před spuštěním REST API a běží na výchozím portu 8080 s nastavením výchozího uživatele.

```
keycloak:
  image: quay.io/keycloak/keycloak:24.0.3
  container_name: keycloak
  environment:
    KC_HOSTNAME: localhost
    KC_HOSTNAME_PORT: 8080
    KC_HOSTNAME_STRICT: false
    KC_HOSTNAME_STRICT_HTTPS: false

    KC_LOG_LEVEL: info
    KC_METRICS_ENABLED: true
    KC_HEALTH_ENABLED: true
    KEYCLOAK_ADMIN: admin
    KEYCLOAK_ADMIN_PASSWORD: admin
  command: start-dev
  ports:
    - 8080:8080
  restart: unless-stopped
```

Keycloak konzole

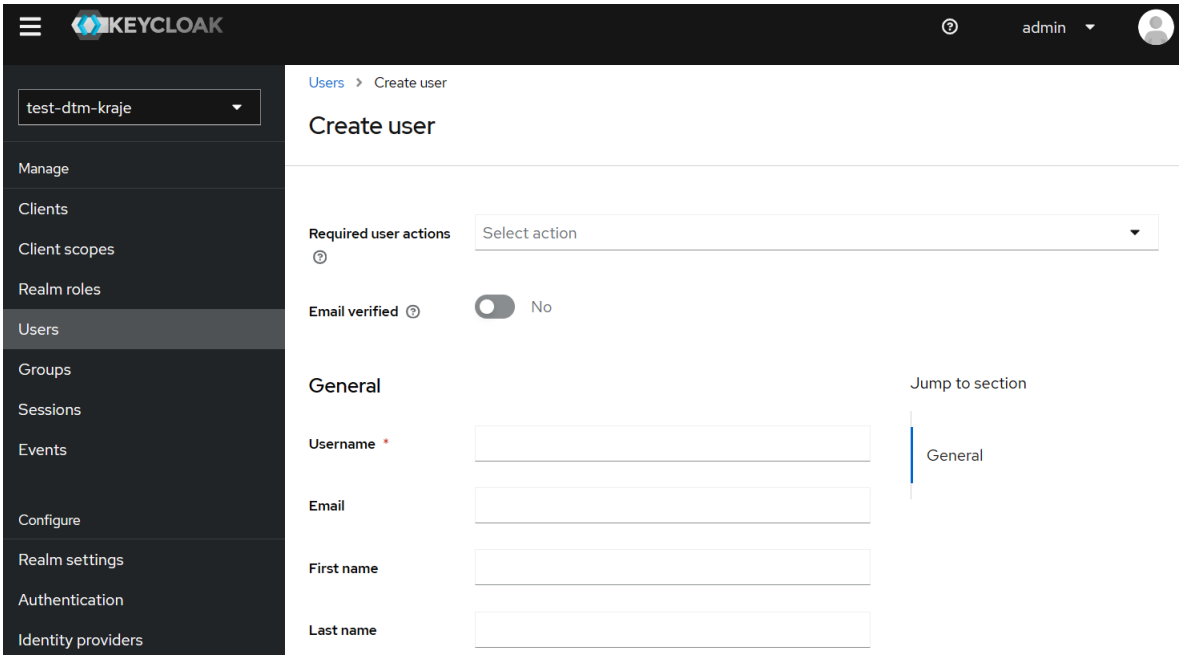
Po nasazení Keycloaku do Docker kontejneru bylo nejprve potřeba vytvořit realm, ve kterém bude nakonfigurován klient pro potřebu tohoto systému. Realm byl pojmenován jako *test-dtm-kraje* a nastaven, aby používal OpenID Connect (OIDC) protokol k ověřování uživatelů.

Následně byl vytvořen klient s názvem *geo-web-control*, který představuje veškerá nastavení pro zabezpečení API a webové aplikace implementovaného distribuovaného systému. Důležité je pro klienta nastavit CORS pravidla včetně validní URI pro přesměrování

po přihlášení a také validní adresy webů, pro které bude povolen přístup. Dále byly vytvořeny tyto role pro jednotlivé skupiny kontrol, které je možné přiřazovat uživatelům:

- **all-controls** – uživatel s tímto oprávněním má povoleno provádět libovolnou skupinu kontrol,
- **data-structure-control** – oprávnění provádět kontroly z kategorie struktury dat,
- **topology-areas** – oprávnění provádět kontroly z kategorie topologických plošných kontrol,
- **topology-detailed-points** – oprávnění provádět kontroly podrobných bodů,
- **topology-points-lines** – oprávnění provádět topologické kontroly bodů a linií.

Pro potřeby tohoto distribuovaného systému nebylo vyžadováno, aby se uživatelé mohli registrovat sami. Požadavek byl, aby bylo možné uživatele vytvořit jednorázově pro každého připojeného klienta v rámci Keycloak realmu. Proto je využito rozhraní Keycloak konzole pro tvorbu a správu uživatelských účtů. Každému uživateli je zde nastaveno uživatelské jméno, heslo, email a systém mu vygeneruje unikátní KeycloakID, dle kterého jsou uživatelé hned vedle uživatelského jména a hesla v rámci systému rozlišování.



Obrázek 24: Přidání uživatele z Keycloak konzole. Zdroj vlastní.

V praxi poté Keycloak zpracovává autentizační požadavky od klientů, ověřuje uživatele a následně vydává JWT tokeny, které umožňují přístup k chráněným zdrojům. JWT tokeny obsahují informace o uživateli a jeho právech, což aplikaci umožňuje autorizovat

uživatele pro přístup k různým službám bez potřeby dalšího přihlášení. Po ověření identity uživatele jsou dále využívány informace o rolích z JWT tokenu pro ověření oprávnění při volání endpointů API konkrétních skupin kontrol. Pro usnadnění získání informací o rolích z tokenu byla do tokenu pomocí mapování přidána položka "GWC-ROLE", do které byly namapovány všechny klientské role daného uživatele.

6.4 REST API

Další klíčovou součástí distribovaného systému je REST API, které bylo implementováno na platformě .NET jako ASP.NET Core Rest API. .NET je rozsáhlá platforma od společnosti Microsoft pro vývoj různých typů softwaru, včetně webů, aplikací, mikroslužeb a atp. ASP.NET Core je nástupcem staršího ASP.NET. Jedná se o otevřený webový framework, který umožňuje tvorbu výkonných webových aplikací a služeb. Framework je navržen tak, aby mohl běžet multiplatformě, včetně Windows, Linuxu a macOS, což umožňuje vývojářům využívat framework bez omezení na operační systém. Tento framework využívá objektově orientovaný jazyk C#. [59], [60]

V obecném popisu implementace distribuovaného systému byla zmíněna možnost synchronního či asynchronního provedení kontroly, které API poskytuje. Synchronní způsob je proveden tak, že klient čeká na odpověď API synchronně, tedy je blokován a čeká na přijetí výstupního protokolu po proběhnutí zpracování veškerých kontrol. Tento způsob se hodí spíše pro použití z desktopové aplikace a pro menší datové soubory. Kontrolovaná data mohou být velkého rozsahu, proto není někdy vhodné synchronně čekat na zpracování, jelikož v případě velkých dat může zpracování trvat i déle než hodinu. Mimo jiné, synchronní způsob zpracování požadavků se nehodí pro webové aplikace. Proto byl implementován také asynchronní způsob zpracování kontrol, kde API poskytuje asynchronní endpointy. V takovém případě uživatel pošle data ke kontrolám a ihned dostane jako odpověď unikátní identifikátor zprávy v systému. Kontrola v systému na pozadí probíhá dál a uživatel se na základě získaného unikátního ID zprávy může zeptat zda je kontrola již zpracována a získat tak výsledek. Aplikace pro rozhraní kontrol totiž asynchronní zprávy ukládá do front, které jsou pojmenovány a používají jako směrovací klíč unikátní ID asynchronní zprávy.

Docker konfigurace

Kontejner pro REST API je založen na obrazu aspnet verze 6 z veřejného repozitáře Dockeru. Obraz je konfigurován v souboru Dockerfile včetně kopírování publikovaného projektu API do kontejneru, nastavení pracovního adrešáře, který port má být otevřen a příkaz, který bude se spuštěním kontejneru proveden.

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS runtime

MAINTAINER silvia.cmilanska@geovap.cz

ENV ASPNETCORE_URLS=http://+:5000

# Copy the published ASP .NET app to the container
COPY ./ErrorControlAPI_publish /ErrorControlAPI

# Set the working directory
WORKDIR /ErrorControlAPI

# Expose port 5000
EXPOSE 5000

# Start the ASP .NET app
ENTRYPOINT ["./ErrorControlAPI"]
```

Kontejner pro REST API je konfigurován v jednom docker-compose souboru spolu se službami Keycloak a databází, na kterých je závislý. Konfigurace obsahuje nastavení názvu kontejneru, cestu k souboru Dockerfile, který definuje obraz, mapování portů, nastavení restartování kontejneru a nastavení pro spuštění až po startu kontejnerů pro Keycloak a PostgreSQL databázi.

```
  error_control_web_api:
    container_name: error_control_web_api
    build:
      context: ./ErrorControlAPI
      dockerfile: Dockerfile
    ports:
      - "5000:5000"
    environment:
      - ASPNETCORE_URLS=http://+:5000
    restart: unless-stopped
    depends_on:
      - db
      - keycloak
```

Konfigurace projektu

Projekt bylo nejprve zapotřebí nastavit, aby se správně připojoval ke službám, na kterých je závislý. A to konkrétně k RabbitMQ, PostgreSQL databázi a Keycloaku. Konfigurace projektu v ASP.NET Core byla provedena pomocí uvedení hodnot ve tvaru klíč:hodnota do souboru *appsettings.json*. Všechny služby běžící v Docker kontejneru mají název hostitele (hostname) adresován pomocí názvu kontejneru.

```
"RabbitMQ": {
  "ConnectionUri":
    "amqp://controlsvc:7a53QO6ltMHv4EOf@rabbitmq:5672/geocontrolsvc",
  "ExchangeName": "error-control",
  "PublishQueueName": "validation",
  "SubscribeResultQueueName": "return-to-api",
  "SubscribeInvalidQueueName": "xml-invalid"
}
```

V předchozím bloku kódu je demonstrována část konfigurace RabbitMQ. V části ConnectionUri je definována uri pro připojení k RabbitMQ uzlu ve tvaru *"amqp://username:password@host:port/virtualhost"*. Ostatní položky obsahují nastavení názvů RabbitMQ front, které API deklaruje, připojuje se k nim, odesílá do nich zprávy a naslouchá příchozím zprávám.

```
"ConnectionStrings": {
  "NpgSqlConnection":
    "XpoProvider=Postgres;Server=db;Port=5432;
    UserID=geo_control_db;Password=geo_control_db;
    Database=geo_control_db;Pooling=true;"
}
```

Tato další ukázka konfigurace zobrazuje nastavení připojení k databázi PostgreSQL pomocí využití ORM knihovny Devexpress XPO.

```
"JWT": {
  "authority": "http://keycloak:8080/realms/test-dtm-kraje",
  "config": "http://keycloak:8080/realms/test-dtm-kraje/
            .well-known/openid-configuration",
  "clientID": "geo-web-control"
}
```

Poslední konfigurace uvedená v souboru appsettings.json zaznamenává url Keycloak realmu, url endpointu pro získání konfigurací a název klienta, což jsou hodnoty potřebné

k ověřování požadavků pomocí Keycloaku.

Implementace RabbitMQ

Připojení REST API k RabbitMQ je provedeno pomocí uri pro připojení, která je specifikována v souboru appsettings.json v sekci RabbitMQ. V následující ukázce kódu je demonstrováno vytvoření připojení pomocí ConnectionFactory a vytvoření kanálu, přes který jsou následně prováděny všechny operace.

```
ConnectionFactory factory = new()
    { Uri = new Uri(_rabbitMqConnectionUri) };

using (IConnection? connection = factory.CreateConnection())
using (IModel? channel = connection.CreateModel())
{
    channel.ExchangeDeclare(exchange: _exchangeName,
                            type: ExchangeType.Direct,
                            durable: true,
                            autoDelete: false,
                            arguments: null);

    channel.QueueDeclare(queue: _publishQueueName,
                         durable: true,
                         exclusive: false,
                         autoDelete: false,
                         arguments: null);

    channel.QueueBind(queue: _publishQueueName,
                      exchange: _exchangeName,
                      routingKey: _publishQueueName,
                      arguments: null);
    ...
}
```

Zároveň je při vytvoření připojení proveden pokus o vytvoření exchange, front a registrace front k exchange. Toto se provede pouze pokud takové objekty v RabbitMQ ještě neexistují, jinak tyto definice nemají žádný efekt. Díky této explicitní definici potřebných objektů RabbitMQ v klientské aplikaci je zajištěno, že vždy budou tyto objekty existovat a nemůže nastat problém s pokusem o připojení k objektu, který neexistuje.

Exchange je vytvořena pomocí metody ExchangeDeclare s parametry: název exchange, což je v tomto případě "error-control", dále typ, který byl nastaven jako Direct Exchange, což znemaná, že používá přímé routování zpráv dle hodnoty směrovacího klíče zprávy.

Další nastavené argumenty jsou durable=true, takže po restartu brokeru nebude exchange smazána, autoDelete=false nastavuje aby po odpojení poslední připojené fronty nebyla exchange smazána. Poslední argument null znamená, že nejsou posílány žádné další volitelné arumenty pro nastavení Exchange.

Fronty se vytvářejí pomocí metody QueueDeclare. Tato metoda má podobné parametry jako metody pro vytvoření Exchange. Výjimkou je parametr exclusive, který nastavuje zda může být fronta používána pouze jedním připojením a po jeho zavření je smazána.

Následně je nutné nově vytvořenou frontu připojit do exchange, která má spravovat směrování zpráv dané fronty. To se provádí metodou QueueBind, kde se specifikuje název fronty, název exchange, směrovací klíč a případně volitelné argumenty.

Asynchronní zpracování zpráv

Při asynchronním zpracování zpráv je vygenerováno GUID pro zprávu, která je poslána do fronty pojmenované "validation", z níž odebírá zprávy aplikace pro validaci XML dat dle XSD schématu. Vygenerované GUID je spolu s dalšími informacemi potřebnými pro kontroly uloženo do headerů RabbitMQ zprávy. Následně je klientovi vráceno pouze GUID zprávy, na kterou se poté může dotázat.

Zpráva je pomocí RabbitMQ publikována voláním metody BasicPublish, která přijímá jako argumenty: název exchange, která směruje danou frontu, hodnotu směrovacího klíče, properties obsahující mimo jiné hlavičky zprávy a nakonec samotná data zprávy, které se posílají jako byte[].

```
channel.BasicPublish(exchange: _exchangeName,
                     routingKey: _publishQueueName,
                     basicProperties: properties,
                     body: body);
```

Další důležitou částí asynchronního zpracování kontrol je metoda IsMessageInResults, která zjišťuje zda je zpráva s daným GUID zprávy přítomna v RabbitMQ. Pro předcházení problémů s neexistující exchange nebo frontou je provedena jejich definice – pokud existují, tak jsou příkazy ignorovány. Při asynchronní variantě mikroslužba rozhraní kontrol vytváří pro výsledky frontu pojmenovanou stejně jako je GUID zprávy a do ní vkládá pouze jedinou zprávu, a to výsledek pro dané GUID. Proto stačí pro zjištění, zda je již výsledek

kontroly dostupný, získat instanci fronty s daným GUID a zjistit, zda obsahuje nějakou zprávu. Viz následující ukázka kódu.

```
channel.ExchangeDeclare(exchange: _exchangeName,
                        type: ExchangeType.Direct,
                        durable: true,
                        autoDelete: false,
                        arguments: null);

QueueDeclareOk? model = channel.QueueDeclare(queue: messageGUID,
                                             durable: true,
                                             exclusive: false,
                                             autoDelete: false,
                                             arguments: null);

bool messageExists = model.MessageCount > 0;
```

Synchronní zpracování zpráv

Při synchronní variantě zpracování zpráv je stejně jako u asynchronní varianty vygenerováno GUID zprávy a zpráva je poslána do systému pomocí metody BasicPublish. Následně je zaregistrováno odebírání zpráv z fronty, která má hodnotu názvu a směrovacího klíče rovnu *"return-to-api"* a čeká se, dokud není přijata výsledná zpráva po zpracování kontroly v systému.

Na následující ukázce kódu je zobrazeno, jak se registruje odebírání zpráv ze specifické fronty. Je vytvořena instance třídy EventingBasicConsumer a následně je pro ni registrováno pomocí metody BasicConsume odebírání zpráv ze specifické fronty a zda budou zprávy automaticky potvrzovány či ne. Při tomto synchronním přístupu bylo důležité nastavit automatické potvrzování na hodnotu false, protože je nejprve potřeba ověřit, zda přijatá zpráva skutečně patří tomuto klientovi.

```
EventingBasicConsumer? resultConsumer = new(channel);

channel.BasicConsume(queue: _subscribeResultQueueName,
                     autoAck: false,
                     consumer: resultConsumer);
```

Logika odebírání zpráv z fronty je zobrazena na následující ukázce kódu. Důležité je vytvoření instance třídy TaskCompletionSource, na které je zavoláno await pro blokování předčasného dokončení metody. Díky tomu je vrácen výsledek až ve chvíli, kdy je nastaven pomocí metody taskCompletionSource.TrySetResult(body), která je provededa až po

získání zprávy z RabbitMQ. To zaručí, že asynchronní metoda neskončí předčasně, než je získán výsledek kontroly, a v klientovi simuluje synchronní průběh čekání na výsledek.

Z hlediska RabbitMQ probíhá získání zprávy tak, že pro instanci EventingBasicConsumer je registrováno naslouchání na událost Recieved. Ve chvíli, kdy je takto přijata zpráva, je nutné zkontrolovat pomocí metody IsGuidEqualToHeaderId, zda má zpráva požadované GUID, tudíž jestli nepatří jinému klientovi. Pokud má správné GUID, tak je odesláno potvrzení o přijetí zprávy a nastaven výsledek. Pokud má jiné GUID než očekávané, znamená to, že patří jinému klientovi. Proto je zpráva odmítnuta voláním metody BasicNack s nastavením, aby byla zařazena zpět do fronty pokud možno nejblíže svému původnímu umístění.

```
TaskCompletionSource<byte[]>? taskCompletionSource = new();
EventingBasicConsumer? resultConsumer = new(channel);

resultConsumer.Received += (model, eventArgs) =>
{
    byte[]? body = eventArgs.Body.ToArray();

    // check if message belongs to this client
    if (IsGuidEqualToHeaderId(eventArgs.BasicProperties.Headers,
                              messageId))
    {
        // withdraw the message
        channel.BasicAck(deliveryTag: eventArgs.DeliveryTag,
                         multiple: false);
        ...
        taskCompletionSource.TrySetResult(body);
    }
    else
    {
        // requeue the unwanted message
        channel.BasicNack(deliveryTag: eventArgs.DeliveryTag,
                          multiple: false,
                          requeue: true);
    }
};
...
byte[]? response = await taskCompletionSource.Task;
```

Následně je demonstrováno jak probíhá ověření, zda má zpráva očekávanou hodnotu GUID. Tuto hodnotu je nutné získat z hlavičky zprávy přijaté z RabbitMQ.

```
private bool IsGuidEqualToHeaderId(IDictionary<string, object> headers,
                                   string guid)
```

```
{
    if (headers.TryGetValue("message-guid", out object? messageId))
    {
        string id = Encoding.UTF8.GetString((byte[])messageId);
        Console.WriteLine($"Received control result with GUID: {id}");
        return messageId is byte[] idBytes
                && Encoding.UTF8.GetString(idBytes).Equals(guid);
    }
    return false;
}
```

Dokumentace endpointů API

V této podkapitole budou popsány kontrolery a poskytované endpointy API. Pro každý endpoint bude zmíněno k čemu slouží a jaké jsou jeho vstupní a výstupní hodnoty. Vstupem všech endpointů je .zip soubor obsahující .xml data.

ZPSAsyncController

Tento kontroler slouží k posílání asynchronních kontrol. Všechny endpointy v kontroleru pro asynchronní komunikaci vracejí GUID zprávy, která byla poslána do systému. Zároveň je u všech endpointů ověřována správnost formátu vstupních dat. A to konkrétně, že nahraný soubor .zip obsahuje platné soubory JVFDTM .xml a volitelně může obsahovat i soubory ve formátu shapefile (.shp, .dbf, .shx).

```
[Authorize(Policy = "all-controls")]
[HttpPost("api/async/davka")]
[Consumes("multipart/form-data")]
public async Task<IActionResult> PerformAllControlsAsync(IFormFile zipInput)
```

- URI: *api/async/davka*
- Pošle data k provedení dávky všech kontrol.
- Díky policy povoleno pouze pro uživatele s rolí *all-controls*.

```
[Authorize(Policy = "data-structure-control")]
[HttpPost("api/async/strukturaDat")]
[Consumes("multipart/form-data")]
public async Task<IActionResult> DataStructureControl(IFormFile zipInput)
```

- URI: *api/async/strukturaDat*
- Pošle data k provedení kontrol struktury dat, které zahrnují kontrolu typu geometrie, souřadnic / nadmořské výšky, zaokrouhlení souřadnic a kontrolu naplnění atributu číselníků.

- Díky policy povoleno pouze pro uživatele s rolí *data-structure-control*.

```
[Authorize(Policy = "topology-detailed-points")]
[HttpPost("api/async/topoPodrobneBody")]
[Consumes("multipart/form-data")]
public async Task<IActionResult> TopologyDetailedPoints(IFormFile zipInput)
```

- URI: *api/async/topoPodrobneBody*
- Pošle data k provedení topologických kontrol podrobných bodů.
- Díky policy povoleno pouze pro uživatele s rolí *topology-detailed-points*.

```
[Authorize(Policy = "topology-points-lines")]
[HttpPost("api/async/topoBodyLinie")]
[Consumes("multipart/form-data")]
public async Task<IActionResult> TopologyPointsAndLines(IFormFile zipInput)
```

- URI: *api/async/topoBodyLinie*
- Pošle data k provedení topologických kontrol bodů a linií, které zahrnují kontrolu úseček nulové délky, objektace, křížení, duplicit, duplicit do min Z, volných konců a blízkých vrcholů.
- Díky policy povoleno pouze pro uživatele s rolí *topology-points-lines*.

```
[Authorize(Policy = "topology-areas")]
[HttpPost("api/async/topoPlochy")]
[Consumes("multipart/form-data")]
public async Task<IActionResult> TopologyAreas(IFormFile zipInput)
```

- URI: *api/async/topoPlochy*
- Pošle data k provedení plošných kontrol.
- Díky policy povoleno pouze pro uživatele s rolí *topology-areas*.

ZPSSyncController

Tento kontroler slouží k posílání dat ke kontrole synchronním způsobem. Všechny endpointy v kontroleru pro synchronní komunikaci vracejí výsledný protokol ve jsko xml soubor. Zároveň je u všech endpointů ověřována správnost formátu vstupních dat stejně jako je tomu u asynchronních kontrol. Endpointy kontroleru jsou stejné jako u asynchronního kontroleru, obsahují pouze jinou uri, která má async nahrazeno za sync. Jako příklad je uveden jeden endpoint pro provedení dávky všech kontrol:

```
[Authorize(Policy = "all-controls")]
[HttpPost("api/sync/davka")]
[Consumes("multipart/form-data")]
public async Task<IActionResult> PerformAllControlsAsync(IFormFile zipInput)
```

- URI: *api/sync/davka*
- Pošle data k provedení dávky všech kontrol.
- Díky policy povoleno pouze pro uživatele s rolí *all-controls*.

ResultsController

ResultsController poskytuje endpointy pro zjištění, zda zpráva se specifickým ID zprávy je již zpracovaná a pro vyzvednutí výsledků. Je to tedy nástroj pro získání výsledků kontrol, které byly odeslány do systému asynchronně.

```
[Authorize]
[HttpPost("api/results/isMessageInQueue")]
public ActionResult<bool> IsSpecifiedMessageInQueue(
                              [FromBody] ResultMessageRequest request)
```

- URI: *api/results/isMessageInQueue*
- Vstupní hodnotou je GUID zprávy v systému.
- Zjišťuje zda je zpráva s daným GUID zprávy přítomna v systému, tedy jestli je už zpracována. Vrací boolean sdělující zda je zpráva přítomna.
- Povoleno pro všechny přihlášené uživatele.

```
[Authorize]
[HttpPost("api/results/getMessage")]
public async Task<IActionResult> GetResultFileFromQueue(
                                     [FromBody] ResultMessageRequest request)
```

- URI: *api/results/getMessage*
- Vstupní hodnotou je GUID zprávy v systému.
- Vrací XML soubor výstupního protokolu kontrol, který je v systému registrován s daným GUID zprávy. Data jsou získána z RabbitMQ fronty.
- Povoleno pro všechny přihlášené uživatele.

ControlController

Slouží pro získání dat o kontrolách přihlášeného uživatele, které jsou zobrazovány ve webové aplikaci.

```
[Authorize]
[HttpGet("api/userControls")]
public async Task<IEnumerable<ControlOutput>> Get()
```

- URI: *api/userControls*

- Vrací všechny informace o provedených či odeslaných kontrolách přihlášeného uživatele.
- Povoleno pro všechny přihlášené uživatele.

ControlTypeController

Kontroler poskytující jednu GET metodu pro získání záznamů z číselníku typu kontrol, které jsou zobrazovány ve webové aplikaci.

```
[Authorize]
[HttpGet("api/controlTypes")]
public async Task<IEnumerable<ControlTypeOutput>> Get()
```

- URI: *api/controlTypes*
- Vrací záznamy z číselníku typu kontrol.
- Povoleno pro všechny přihlášené uživatele.

StatesController

Tento kontroler poskytuje jednu GET metodu pro získání záznamů z číselníku stavů výsledku kontrol, které jsou zobrazovány ve webové aplikaci.

```
[Authorize]
[HttpGet("api/states")]
public async Task<IEnumerable<StateOutput>> Get()
```

- URI: *api/states*
- Vrací záznamy z číselníku stavů výsledku kontrol.
- Povoleno pro všechny přihlášené uživatele.

ResultFileController

Kontroler pro získání dat o specifickém souboru, který byl zpracován systémem a uložen do databáze.

```
[Authorize]
[HttpPost("api/resultFile")]
public async Task<IActionResult> FetchResultFileById(ResultFileRequest request)
```

- URI: *api/resultFile*
- Vstupní hodnotou je GUID zprávy v systému, která identifikuje výsledný soubor.
- Vrací entitu ResultFileOutput obsahující název a data výsledného xml protokolu. Data získává z databáze na rozdíl od endpointu v kontroleru ResultsController, kde jsou data získána přímo z RabbitMQ fronty.
- Povoleno pro všechny přihlášené uživatele.

6.5 Validace XML dle XSD

Aplikace pro validaci XML souboru dle komplexního XSD schématu funguje jako mikroslužba v distribuovaném systému. Jedná se o konzolovou aplikaci napsanou v jazyce C# na platformě .NET. Hlavní logikou aplikace je připojení k RabbitMQ, přijetí zprávy z fronty pojmenované *"validation"*, rozzipování dat a provedení validace .xml souboru. Pokud je soubor validní, tak aplikace odešle data k dalšímu zpracování do fronty *"xml-to-control"*. Pokud data nejsou validní, tak mohou nastat dvě situace. Buď se jedná o asynchronní variantu zpracování – v takovém případě je vytvořena fronta pojmenovaná dle GUID přijaté zprávy a jsou do ní odeslána data. Anebo se jedná o synchronní variantu zpracování – v takovém případě je zpráva odeslána do fronty *"xml-invalid"*.

Na následující ukázce kódu je možné vidět nastavení aplikace tak, aby běžela v nekonečném naslouchání příchozím zprávám do RabbitMQ fronty a byla zastavena pouze při přijetí ukončujícího signálu například pomocí klávesové zkratky Ctrl+C.

```
CancellationTokenSource? cancelTokenSource = new();
Console.CancelKeyPress += (sender, eventArgs) =>
{
    // Handle Ctrl+C or other termination signals
    eventArgs.Cancel = true;
    cancelTokenSource.Cancel();
};

RabbitMQValidationClient client = new(conUri);
client.ConnectToRabbitMQ(cancelTokenSource);

...

while (!cancelTokenSource.IsCancellationRequested)
{ Thread.Sleep(1000); }
```

Logika vytvoření připojení a kanálu pro práci s RabbitMQ, stejně jako logika odebírání a publikování zpráv je provedena obdobně jako bylo popsáno v kapitole o REST API.

Docker konfigurace

Kontejner pro mikroslužbu provádějící validaci XML dat je stejně jako API založen na obrazu aspnet verze 6 z veřejného repozitáře Dockeru. Obraz je konfigurován v souboru Dockerfile včetně kopírování dat publikovaného projektu Validate_XML_XSD_Client do

kontejneru, nastavení pracovního adrešáře a příkaz, který bude se spuštěním kontejneru proveden.

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS runtime

MAINTAINER silvia.cmilanska@gmail.com

# Copy the published .NET Core app to the container
COPY ./Validate_XML_XSD_Client_publish /Validate_XML_XSD_Client

# Set the working directory
WORKDIR /Validate_XML_XSD_Client

# Start the .NET Core app
CMD ["./Validate_XML_XSD_Client"]
```

Konfigurace docker-compose souboru obsahuje nastavení pracovního adresáře kontejneru, cestu k souboru Dockerfile a nastavení restartování při pádu.

```
  validate_xml_xsd_client:
    build:
      context: ./Validate_XML_XSD_Client
      dockerfile: Dockerfile
    restart: unless-stopped
```

Implementace validace

Jak již bylo popsáno v obecném představení chování této aplikace – po přijetí zprávy z RabbitMQ fronty pojmenované *"validation"* jsou data ze zipu extrahována. Poté je zjištěna verze přijatého JVF DTM .xml souboru a dle verze je vrácena cesta ke korektnímu schématu XSD, které bude použito při validaci. Pokud pro danou verzi nebyla nalezena shoda a vrácena cesta ke schématu, znamená to, že se jedná o nepodporovaný formát a data jsou nevalidní. Pokud je verze validní, tak se pokračuje na samotnou validaci XML dat. Pro implementaci validace byly využity systémové knihovny .NET pro XML a XML Schema.

Nejprve je pomocí metody SchemaSetProvider.GetXmlSchemaSet načteno komplexní XML schema do objektu XmlSchemaSet. Poté je provedena samotná validace voláním metody ObjectValidator.Validate, která přijímá jako hodnoty argumentů řetězec xml souboru k validaci a načtený XmlSchemaSet.

```
private bool PerformValidation()
{
```

```
    string schemaPath = GetXsdSchemaPath();
    if (string.IsNullOrEmpty(schemaPath))
    {
        _validationResult = "Unsupported JVF version.";
        return false;
    }

    XmlSchemaSet schemas = SchemaSetProvider.GetXmlSchemaSet(schemaPath);
    ValidationResult result = ObjectValidator.Validate(XmlFileString,
                                                       schemas);

    _validationResult = result.ToString();
    Console.WriteLine(_validationResult);

    return result.ValidationState == ValidationState.Valid;
}
```

Jelikož je XSD schéma velice komplexní, slouží jako hlavní vstupní bod pro načtení celého schématu soubor *index_data.xsd*. Tento soubor obsahuje definice všech oborů názvů a objektů, které jsou ve schématu zahrnuty, a zároveň uvádí cesty k dalším XSD souborům, které obsahují specifické definice těchto prvků. V následující ukázce kódu je zobrazeno jak probíhá načtení a zkompilování celého XSD schematu do objektu XmlSchemaSet. Protože vstupní index soubor obsahuje mnoho referencí na další soubory a obory názvů, musí být pro správné zpracování do instance objektu XmlSchemaSet přidán XmlUrlResolver.

```
public static XmlSchemaSet GetXmlSchemaSet(string xsdIndexPath)
{
    XmlSchemaSet? schemas = new();
    schemas.XmlResolver = new XmlUrlResolver();
    schemas.Add(null, xsdIndexPath);
    schemas.Compile();
    return schemas;
}
```

Vstupem samotné validace je poté XML dokument jako řetězec a sada kompilovaných schémat. Během validace jsou sbírány informace o chybách pomocí delegátní funkce, která je volána vždy, když validátor narazí na chybu. Sbírané informace zahrnují jméno objektu (element, atribut), zprávu o chybě a informace o umístění chyby v dokumentu.

```
docToValidate.Validate(schemas, (xmlObject, eventArgs) =>
{
    XObject? xObject = xmlObject as XObject;
```

```
    IXmlLineInfo? lineInfo = xObject as IXmlLineInfo;
    string locationInfo = lineInfo.HasLineInfo() ?
                          $"Line {lineInfo.LineNumber}" : "Line unknown";
    string objectName = "Unknown object";

    if (xObject is XElement element)
    {
        objectName = $"Element {element.Name.LocalName}";
    }
    else if (xObject is XAttribute attribute)
    {
        objectName = $"Attribute {attribute.Name.LocalName} " +
                     $"in element {attribute.Parent.Name.LocalName}";
    }

    string message = $"{objectName}: " +
                     $"{eventArgs.Message} ({locationInfo})";
    validationMessages.Add(
        new ValidationMessage(message, eventArgs.Severity));
});
```

6.6 Rozhraní pro kontroly

Aplikace v roli rozhraní pro kontroly je poslední mikroslužbou v implementovaném distribuovaném systému. Aplikace je také napsána v programovacím jazyce C# .NET. Stejně jako je tomu u předchozí mikroslužby pro validaci, jedná se o konzolovou aplikaci, která běží dokud nepřijme přerušovací signál.

Docker konfigurace

Konfugurace docker kontejneru pro mikroslužbu rozhraní geodetických kontrol je obdobná jako u aplikace pro validaci XML dat. Je využívám stejný obraz jako u validace či API, obdobně se v Dockerfile nastavuje kopírování publikovaného projektu do kontejneru, pracovní adresář a příkaz pro spuštění aplikace.

Kontejner pro mikroslužbu provádějící validaci XML dat je stejně jako API založen na obrazu aspnet verze 6 z veřejného repozitáře Dockeru. Obraz je konfigurován v souboru Dockerfile včetně kopírování dat publikovaného projektu Validate_XML_XSD_Client do kontejneru, nastavení pracovního adrešáře a příkaz, který bude se spuštěním kontejneru proveden.

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS runtime

MAINTAINER silvia.cmilanska@gmail.com

# Copy the published .NET Core app to the container
COPY ./ErrorControlMQInterface_publish /ErrorControlMQInterface

# Set the working directory
WORKDIR /ErrorControlMQInterface

# Start the .NET Core app
ENTRYPOINT ["./ErrorControlMQInterface"]
```

Konfigurace docker-compose je stejná jako u kontejneru pro validaci s rozdílem jin=ého názvu služby a jiné cesty k pracovnímu adresáři.

```
  error_control_mq_interface:
    build:
      context: ./ErrorControlMQInterface
      dockerfile: Dockerfile
    restart: unless-stopped
```

Implementace aplikace

Aplikace se nejprve připojuje k RabbitMQ, obdobně jako ve validační mikroslužbě, pomocí RabbitMQ klientské knihovny pro C#.

Následující ukázka kódu demonstruje zaregistrování instance třídy EventingBasicConsumer pro odebírání zpráv z fronty *"xml-to-control"*. Důležité je zde zmínit, že na rozdíl od ukázky metody BasicConsume v API je zde parametr autoAck nastaven na hodnotu true, aby došlo k automatickému potvrzení každé přijaté zprávy.

```
EventingBasicConsumer? consumer = new(channel);
channel.BasicConsume(queue: "xml-to-control",
                     autoAck: true,
                     consumer: consumer);
```

Po připojení k RabbitMQ aplikace čeká na příchozí zprávy z fronty *"xml-to-control"*. Po přijetí je jsou zip data zprávy rozbalena, z dat je načten JVFDTM XML soubor a případně shapefile soubory, pokud jsou v zipu obsaženy. Dále jsou také přečtena data z hlavičky RabbitMQ zprávy nesoucí informace o typu volání (sync/async) a o skupině kontrol, která má být provedena.

Ukázka kódu uvedena pod tímto textem popisuje jak jsou volány kontroly pomocí metod poskytovaných připojeným .dll souborem s algoritmy geodetických kontrol. Nejprve jsou načteny jednotlivé geodetické prvky z kontrolovaného souboru JVFDTM a poté již mohou být spuštěny kontroly na načtených geodetických prvcích s využitím konfiguračního souboru dle typu požadované skupiny kontrol a případně s vymezením kontrolovaného území podle načtených dat z shapefile souboru. Po dokončení kontroly je do hlavičky RabbitMQ zprávy uložena informace o počtu chyb, která je využívána ve webové aplikaci. Nakonec je načten řetězec formující výsledný XML protokol proběhlých kontrol, který je datovým výstupem distribuovaného systému.

```
ErrorControl control = new();
List<GeoElement> elements = control.ReadGeoElementsFromFile(XmlPath);

ResultXmlOutput errors = control.ExecuteControlBatch(
                                    parFilePath,
                                    ShpPath,
                                    elements);

eventArgs.BasicProperties.Headers.Add("error-count",
                                     $"{errors.ErrorCount}");

string controlProtocol = control.FetchResultXmlString();
```

Poslední ukázka kódu zobrazuje odeslání výsledného XML protokolu ve formě byte[] pomocí RabbitMQ zprávy. Pokud uživatel zvolil asynchronní variantu zpracování kontrol, tak je vytvořena nová RabbitMQ fronta, jejíž název i směrovací klíč mají hodnotu rovnu GUID RabbitMQ zprávy, která doručila požadavek na kontrolu. Do nově vytvořené fronty je poslána zpráva obsahující výsledný XML protokol. Této zprávě je nastavena expirace na 24 hodin, tzn. že pokud si uživatel nevyžádá výsledek s daným GUID zprávy do 24 hodin od jejího zpracování, tak je zpráva z fronty odstraněna. V případě, že se nejedná o variantu asynchronní, ale bylo vyžádáno provedení synchronní kontroly, tak je výsledek odeslán do fronty *"return-to-api"*. Novým zprávám v této frontě naslouchá REST API a po přijetí zprávy doručí výsledek uživateli do aplikace, přes kterou kontrolu vyžádal.

```
if (callType.Equals("async"))
{
    string messageId = ReadMessageIdFromHeaders(headers);

    channel.QueueDeclare(queue: messageId,
```

```
                        durable: true,
                        exclusive: false,
                        autoDelete: false,
                        arguments: null);

    channel.QueueBind(queue: messageId,
                      exchange: "error-control",
                      routingKey: messageId,
                      arguments: null);

    // message expires after 24h
    deliveryEventArgs.BasicProperties.Expiration = "86400000";
    channel.BasicPublish(
                  exchange: "error-control",
                  routingKey: messageId,
                  basicProperties: deliveryEventArgs.BasicProperties,
                  body: resultFile);
}
else
{
    channel.BasicPublish(
                  exchange: "error-control",
                  routingKey: "return-to-api",
                  basicProperties: deliveryEventArgs.BasicProperties,
                  body: resultFile);
}
```

6.7 Webová aplikace

Pro implementaci uživatelského rozhraní, které uživateli umožňuje interakci s distribuovaným systémem, byla vyvinuta webová aplikace s využitím technologií React a TypeScript. Pro sestavení této aplikace byl použit nástroj Vite. Styly pro grafické komponenty webové aplikace byly vytvořeny pomocí css knihovny Tailwind CSS.

React je JavaScriptová knihovna umižňující tvorbu uživatelských rozhraní pomocí nezávislých komponent, které jsou schopny spravovat vlastní stav. TypeScript je rozšíření JavaScriptu o statické typování, což zlepšuje produktivitu i bezpečnost vývoje. Tato kombinace technologií – React, TypeScript a Vite – tvoří silný základ pro implementaci uživatelského rozhraní webové aplikace, která slouží jako vstupní bod do distribuovaného systému pro geodetické kontroly. Umožňuje rychlý vývoj, robustní architekturu a efektivní správu jak uživatelského rozhraní, tak i stavu aplikace. [61]

Docker konfigurace

Docker kofigurace pro nasazení webové aplikace v Docker kontejneru obsahuje Dockerfile s konfigurací obrazu, který je založen na oficiálním Node obrazu verze 18. Poté kopíruje do kontejneru soubory package.json a instaluje z něj příkazem npm install všechny potřebné závislosti. Dále je nastaveno, který port bude otevřen a příkaz spouštějící aplikaci.

```
FROM node:18

MAINTAINER silvia.cmilanska@gmail.com

WORKDIR /geo-control-fe

COPY package*.json ./

# Install all package dependencies specified in package.json
RUN npm install

COPY . .

EXPOSE 80

ENV VITE_PORT=80

CMD ["npm", "run", "dev"]
```

Soubor docker-compose.yaml obsahuje konfiguraci pracovního adresáře projektu, název kontejneru, mapování portů a volumes pro přístup k datům kontejneru z lokálního zařízení.

```
frontend:
  build: ./geo-control-fe
  container_name: frontend
  ports:
    - "80:80"
  environment:
    - VITE_PORT=80
  volumes:
    - ./geo-control-fe:/geo-control-fe
    - /geo-control-fe/node_modules
  restart: unless-stopped
```

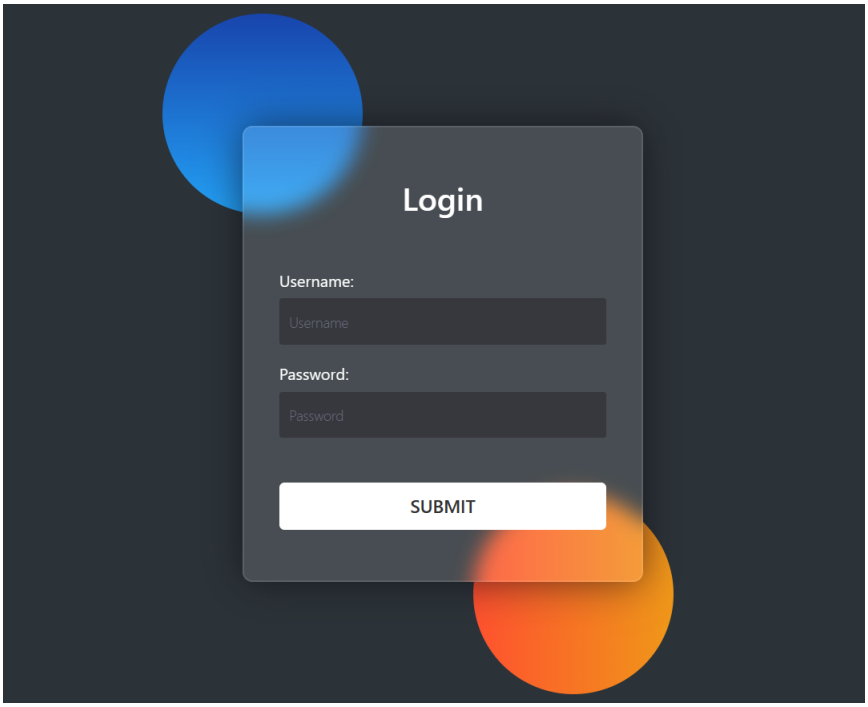
Dále webová aplikace v kontejneru fungovala správně bylo nutné správně nastavit konfigurační soubor vite.config.ts, který nastavuje projekt Vite pro vývoj aplikací v Reactu. Server je konfigurován tak, aby naslouchal na všech síťových rozhraních pomocí adresy

0.0.0.0 a běžel na portu 80, což je standardní port pro HTTP. To umožňuje přístup k vývojovému serveru přímo pomocí *http://localhost* bez nutnosti specifikovat číslo portu. Konfigurace HMR (Hot Module Replacement) specifikuje, že klient, v tomto případě prohlížeč, se má připojit k HMR serveru na portu 80 pod hostitelem localhost.

```
export default defineConfig({
  plugins: [react()],
  server: {
    host: '0.0.0.0',
    port: 80,
    hmr: {
      clientPort: 80,
      host: 'localhost',
    }
  }
});
```

Uživatelské rozhraní

Vstupním bodem do webové aplikace je přihlašovací obrazovka, kde zadáním korektní kombinace přihlašovacího jména a hesla je uživatel ověřen pomocí služby Keycloak. V případě špatných přihlašovacích údajů nebo jiné chyby je pod formulářem zobrazena chybová hláška.

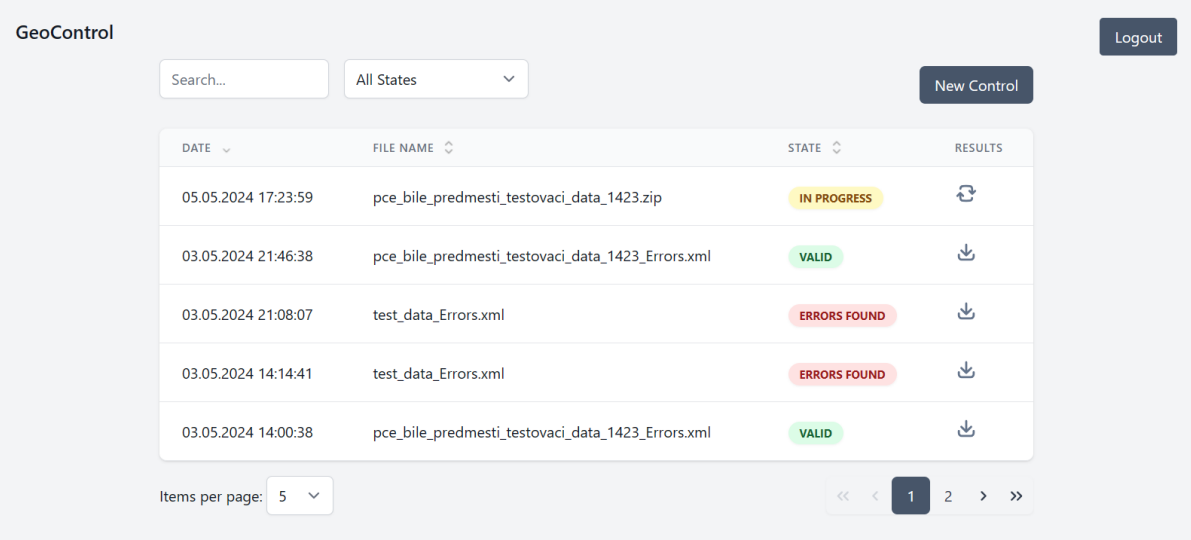


Obrázek 25: Přihlašovací formulář. Zdroj vlastní.

Po úspěšném přihlášení je uživatel přesměrován na hlavní stránku webové aplikace, která obsahuje veškeré ovládací prvky a zobrazuje data kontrol patřící přihlášenému uživateli. Hlavním prvkem stránky je tabulka obsahující seznam provedených geodetických kontrol uživatelem. Pro každou položku seznamu je zobrazeno datum vytvoření kontroly, název souboru, který byl odeslán ke kontrole, stav kontroly a tlačítko pro stažení výsledku.

Horní část stránky nad tabulkou obsahuje textové pole pro vyhledávání záznamů v tabulce dle libovolného sloupce nebo výběr poskytující filtrování záznamů podle stavu. V levé horní části stránky je tlačítko *Logout*, pomocí kterého se může uživatel z aplikace odhlásit a také tlačítko *New Control*, pomocí kterého může uživatel poslat data ke kontrole.

Dolní část stránky pod tabulkou poskytuje vlevo filtr počtu položek tabulky, které budou zobrazeny na stránce. A v pravé spodní části pod tabulkou se nachází ovládání stránkování, pomocí kterého může uživatel zobrazit jak konkrétní číslo stránky seznamu, tak i následující, předchozí, první nebo poslední stránku seznamu kontrol.



Obrázek 26: Uživatelské rozhraní systému. Zdroj vlastní.

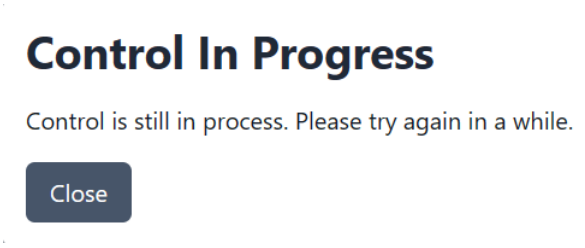
Položka tabulky kontrol se může nacházet v jednom ze 4 stavů:

- **VALID** – stav, kdy kontrola geodetických dat proběhla bez nalezení chyb. Data jsou validní.
- **IN PROGRESS** – stav, kdy byla data odeslána ke kontrole a ještě nebyl získán výsledek. V tomto stavu se může uživatel pokusit načíst výsledek kontroly. Pokud kontrola již doběhla, tak se uživateli zobrazí informace o výsledku, stáhne se pro-

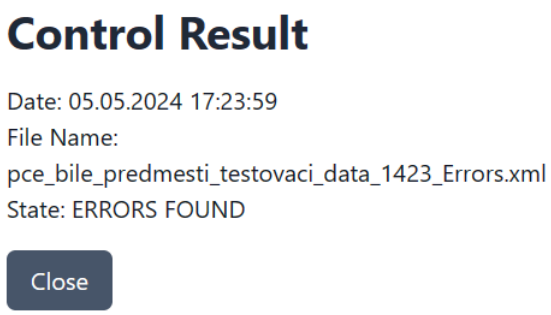
tokol kontrol a položka v tabulce přejde dle výsledku do jednoho ze stavů *VALID*, *WARNING* nebo *ERRORS FOUND*. Pokud zpracování kontroly ještě neskončilo, je uživateli zobrazeno oznámení, že je kontrola stále v procesu a stav položky v tabulce se nezmění.

- **WARNING** – stav, kdy kontrola geodetických dat nalezla chyby v datech, které jsou typu varování, né error.
- **ERRORS FOUND** – stav, kdy kontrola geodetických dat nalezla chyby typu error.

Na obrázku 27 je zobrazena podoba modálního okna, které je uživateli prezentováno ve chvíli, kdy se pokusí získat výsledek položky kontrol ve stavu *IN PROGRESS* a zpracování dané kontroly ještě nebylo dokončeno. V případě, že by zpracování již bylo dokončeno, tak se uživateli stáhne výsledný XML protokol a zobrazí se mu dialog v modálním okně, který je zobrazen na obrázku 28.

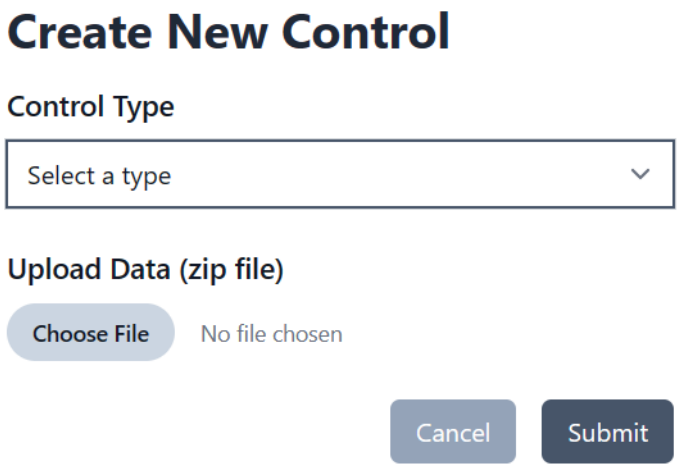


Obrázek 27: Dialog nezpracované kontroly.
Zdroj vlastní.



Obrázek 28: Dialog dokončené kontroly.
Zdroj vlastní.

Obrázek 29 zobrazuje podobu modálního formuláře, který je uživateli zobrazen po kliknutí na tlačítko *New Control*. Tento formulář slouží pro vybrání typu kontroly, kterou chce uživatel provést nad svými geodetickými daty a pro nahrání dat pro kontrolu. Validní data jsou ve formátu .zip obsahující JVFDTM XML soubor a případně shapefile soubory k vymezení oblasti kontroly. Jiné datové formáty nejsou podporovány.



Obrázek 29: Formulář pro odeslání dat ke kontrole. Zdroj vlastní.

Po odeslání dat pomocí tohoto formuláře je v tabulce kontrol uživatele přidána položka s názvem kontrolovaného souboru a stavem *IN PROGRESS*. Výsledek kontroly si může uživatel poté vyžádat pomocí kliknutí na tlačítko u záznamu kontroly v tabulce ve sloupci "Results". Podoba JVFDTM XML souboru a výsledného XML protokolu je popsána v kapitole 5.3.1.

ZÁVĚR

Cílem této diplomové práce bylo navrhnout a implementovat distribuovaný systém pro efektivní a spolehlivou správu kontrol geodetických dat. Mezi požadované vlastnosti systému patřilo splnění vlastností a principů distribuovaných systémů, aby byl systém postaven na platformě .NET, dále byla požadována implementace komunikace s využitím protokolu AMQP, kontejnerizace s využitím technologie Docker a vytvoření uživatelského rozhraní pro zadání dat pro kontrolu a zobrazení výsledků. Všechny výše zmíněné cíle byly během implementace splněny díky důkladnému návrhu systému a jeho dodržení při implementaci.

Výsledný distribuovaný systém tvoří hybridní zprávově orientovanou a mikroservisní architekturu. Veškerá komunikace v systému je zajištěna díky službě RabbitMQ pomocí protokolu AMQP, který řídí doručování zpráv do specificky vytvořených front. Aby RabbitMQ broker nepředstavoval jediný bod selhání, byl nakonfigurován a pomocí Dockeru nasazen jako cluster. Aplikace řídící důležité funkční části distribuovaného systému byly implementovány jako mikroslužby na platformě .NET. Konkrétně se jedná o REST API, které poskytuje endpointy pro provedení jednotlivých skupin kontrol nebo pro získání dat dle potřeb webové aplikace. API využívá pro ukládání dat PostgreSQL databázi a službu Keycloak pro ověřování a správu uživatelů. Druhou mikroslužbou v systému je aplikace pro validaci XML dat dle komplexního XSD schématu. A poslední mikroslužbou je aplikace ve funkci rozhraní geodetických kontrol, která zajišťuje provedení kontrolních algoritmů a předává výsledky zpět do systému. Tyto mikroslužby se připojují k RabbitMQ a komunikují mezi sebou pomocí zasílání a přijímání zpráv ze specifických front s využitím protokolu AMQP. Webová aplikace implementovaná pomocí technologií React, TypeScript a Vite zajišťuje uživatelům intuitivní interakci se systémem, umožňuje odesílání dat ke kontrole a sledování výsledků již proběhlých kontrol.

Každá z popsaných částí je nasazena v Docker kontejnerech, díky čemuž je systém snadno přenositelný, škálovatelný a jeho komponenty jsou v případě potřeby snadno nahraditelné. Díky těmto vlastnostem a implementaci RabbitMQ clusteru byla splněna i podmínka dodržení vlastností distribuovaných systémů.

Praktický výstup této práce je využíván pro kontrolu geodetických dat před jejich importem do informačního systému v rámci projektu Digitální Technické Mapy ČR. Systém

by bylo možné do budoucna rozšířit o další funkcionality uživatelského rozhraní, zlepšení způsobu nasazování jednotlivých částí systému a zlepšení výkonu při práci s velkými objemy dat.

POUŽITÁ LITERATURA

[1] TANENBAUM, Andrew S. A., VAN STEEN, Maarten. *Distributed systems: principles and paradigm.* 2nd ed. Pearson Prentice Hall, 2007. ISBN 0-13-239227-5.

[2] TANENBAUM, Andrew S. A., VAN STEEN, Maarten. *Distributed systems.* 4th ed. Maarten van Steen, 2023. ISBN 978-90-815406-4-3.

[3] TOURON, Manfred. Centralized vs Decentralized vs Distributed Systems. *Berty: Berty Technologies* [online]. Francie, Paříž: Berty Technologies. 20. 6. 2019 [cit. 2024-05-07]. Dostupné z: `https://berty.tech/blog/decentralized-distributed-centralized/`.

[4] SAMBEEK, Bas Van. Why decentralization? *IOTA News | Explore breaking News about IOTA and Shimmer* [online]. 20. 5. 2020 [cit. 2024-05-07]. Dostupné z: `https://iota-news.com/why-decentralization-part-i/`.

[5] cryptotrekking.com. Centralized, Decentralized, and Distributed Systems. *Home: cryptotrekking.com* [online]. 3. 7. 2023 [cit. 2024-05-07]. Dostupné z: `https://cryptotrekking.com/comparison-centralized-decentralized-and-distributed-systems/`.

[6] StormGain. Decentralised vs distributed systems | StormGain *An All-in-One Cryptocurrency App* [online]. 17. 5. 2023 [cit. 2024-05-07]. Dostupné z: `https://stormgain.com/blog/difference-between-decentralised-and-distributed-systems`.

[7] Educative. Characteristics of Distributed Systems. *Educative: AI-Powered Interactive Courses for Developers* [online]. USA, Bellevue, Washington: Educative, Inc., © 2024 [cit. 2024-05-07]. Dostupné z: `https://www.educative.io/courses/scalable-data-pipelines-kafka/characteristics-of-distributed-systems`.

[8] SAKALLI, Duran. Key Characteristics of Distributed Systems. *Medium* [online]. USA, San Francisco: Medium, 9. 10. 2023 [cit. 2024-05-07]. Dostupné z: `https://medium.com/@sakalli.duran/key-characteristics-of-distributed-systems-0cc6e3ee08d3`.

[9] Nitika. Characteristics of Distributed System. *Code 360 by Coding Ninjas* [online]. Indie, Noida: Info Edge, 27. 3. 2024 [cit. 2024-05-07]. Dostupné z: https://www.naukri.com/code360/library/characteristics-of-distributed-system.

[10] annieahujaweb2020. GeeksforGeeks: Types of Transparency in Distributed System. *GeeksforGeeks: a computer science portal for geeks* [online]. Indie, Uttar Pradesh, Noida: GeeksforGeeks, 1. 6. 2022 [cit. 2024-05-07]. Dostupné z: https://www.geeksforgeeks.org/types-of-transparency-in-distributed-system/.

[11] GeeksforGeeks. GeeksforGeeks: Types of Transparency in Distributed System. *GeeksforGeeks: a computer science portal for geeks* [online]. Indie, Uttar Pradesh, Noida: GeeksforGeeks, 2. 12. 2022 [cit. 2024-05-07]. Dostupné z: https://www.geeksforgeeks.org/difference-between-system-architecture-and-software-architecture/.

[12] LASITH, Amanda. Distributed System Architecture Explained. *Medium* [online]. USA, San Francisco: Medium, 20. 10. 2023 [cit. 2024-05-07]. Dostupné z: https://medium.com/@lasithamanda1999/distributed-system-architecture-explained-5e43a65ff4ad.

[13] SUGATHADASA, Keet Malin. Distributed System Architectures and Architectural Styles. *Keet Malin Sugathadasa* [online]. Sri Lanka, Moratuwa: University of Moratuwa, 29. 12. 2022 [cit. 2024-05-07]. Dostupné z: https://keetmalin.wixsite.com/keetmalin/post/2017/09/27/distributed-system-architectures-and-architectural-styles.

[14] MOHANAN, Remya. What Are Distributed Systems? Architecture Types, Key Components, and Examples. *Business and Industry News, Analysis and Expert Insights: Spiceworks* [online]. UK, London: Spiceworks Inc., 12. 1. 2022 [cit. 2024-05-07]. Dostupné z: https://www.spiceworks.com/tech/cloud/articles/what-is-distributed-computing/.

[15] PEIRIS, Imesh. Distributed System Architecture. *Medium* [online]. USA, San Francisco: Medium, 21. 11. 2023 [cit. 2024-05-07]. Dostupné z: https://medium.com/@t.i.tpeeriya/distributed-system-architecture-26b3bc03df4d.

[16] ZETLLER, Kev. What is a distributed system?. *Collaboration software for software, IT and business teams* [online]. Austrálie, Sydney: Atlassian, © 2024 [cit. 2024-05-07]. Dostupné z: https://www.atlassian.com/microservices/microservices-architecture/distributed-architecture.

[17] PAGANINI, Catherine. Primer: Understanding Software and System Architecture. *LinkedIn* [online]. USA, Kalifornie, 18. 12. 2019 [cit. 2024-05-07]. Dostupné z: https://www.linkedin.com/pulse/primer-understanding-software-system-architecture-catherine-paganini/.

[18] RICHMAN, Jeffrey. What are distributed architectures. *Real-time ETL: Estuary* [online]. USA, New York, 27. 6. 2023 [cit. 2024-05-07]. Dostupné z: https://estuary.dev/distributed-architecture/.

[19] Alokai. Microservices in eCommerce Explained. *Frontend as a Service for composable commerce: Alokai* [online]. USA, San Francisco, 23. 8. 2023 [cit. 2024-05-07]. Dostupné z: https://alokai.com/blog/microservices.

[20] OZKAYA, Mehmet. Microservices Architecture for Enterprise Large-Scaled Application. *Medium* [online]. USA, San Francisco: Medium, 17. 2. 2023 [cit. 2024-05-07]. Dostupné z: https://medium.com/design-microservices-architecture-with-patterns/microservices-architecture-for-enterprise-large-scaled-application-825436c9a

[21] annieahujaweb2020. GeeksforGeeks: What is RPC Mechanism in Distributed System? *GeeksforGeeks: a computer science portal for geeks* [online]. Indie, Uttar Pradesh, Noida: GeeksforGeeks, 19. 3. 2022 [cit. 2024-05-07]. Dostupné z: https://www.geeksforgeeks.org/what-is-rpc-mechanism-in-distributed-system/.

[22] annieahujaweb2020. GeeksforGeeks: Stub Generation in Distributed System. *GeeksforGeeks: a computer science portal for geeks* [online]. Indie, Uttar Pradesh, Noida: GeeksforGeeks, 18. 3. 2024 [cit. 2024-05-07]. Dostupné z: https://www.geeksforgeeks.org/stub-generation-in-distributed-system/?ref=lbp.

[23] SHENOY, Prashant. *CMPSCI 677: Distributed and Operating Systems* [online]. UMass Amherst, 2019 [cit. 2024-05-07]. Dostupné z: https://lass.cs.umass.edu/~shenoy/courses/spring19/lectures/Lec10_notes.pdf.

[24] GeeksforGeeks. GeeksforGeeks: MPI - Distributed Computing made easy. *GeeksforGeeks: a computer science portal for geeks* [online]. Indie, Uttar Pradesh, Noida: GeeksforGeeks, 19. 9. 2023 [cit. 2024-05-07]. Dostupné z: `https://www.geeksforgeeks.org/mpi-distributed-computing-made-easy/`.

[25] GILLIS, Alexander S. What is Message Passing Interface (MPI)? *Enterprise Desktop Information, News and Tips from TechTarget* [online]. USA, Massachusetts: TechTarget, Inc., červenec 2022 [cit. 2024-05-07]. Dostupné z: `https://www.techtarget.com/searchenterprisedesktop/definition/message-passing-interface-MPI`.

[26] thesunpandey. Message Queues: System Design. *GeeksforGeeks: a computer science portal for geeks* [online]. Indie, Uttar Pradesh, Noida: GeeksforGeeks, 5. 1. 2024 [cit. 2024-05-07]. Dostupné z: `https://www.geeksforgeeks.org/message-queues-system-design/`.

[27] JOHANSSON, Lovisa. What is Message Queuing? *CloudAMQP: Queue starts here.* [online]. Švédsko, Stockholm: CloudAMQP, 1. 3. 2023 [cit. 2024-05-07]. Dostupné z: `https://www.cloudamqp.com/blog/what-is-message-queuing.html`.

[28] RabbitMQ. AMQP 0-9-1 Model Explained. *RabbitMQ: One broker to queue them all* [online]. UK, London: CloudAMQP, © 2024 [cit. 2024-05-07]. Dostupné z: `https://www.rabbitmq.com/tutorials/amqp-concepts`.

[29] RabbitMQ. Broker Semantics. *RabbitMQ: One broker to queue them all* [online]. UK, London: CloudAMQP, © 2024 [cit. 2024-05-07]. Dostupné z: `https://www.rabbitmq.com/docs/semantics`.

[30] Docker overview. *Docker Documentation* [online]. USA, California: Docker Inc., 17. 4. 2024 [cit. 2024-05-07]. Dostupné z: `https://docs.docker.com/get-started/overview/`.

[31] FREEMAN, Adam. *Essential Docker for ASP.NET Core MVC.* Imprint: Apress, 2017. ISBN 9781484227787.

[32] NICKOLOFF, Jeff. *Docker in Action. 1.* United States of America: Manning Publications Co., 2016. ISBN 9781633430235.

[33] What Is Docker? *IBM - United States* [online]. USA, New York, 2. 4. 2024 [cit. 2024-05-07]. Dostupné z: https://www.ibm.com/topics/docker.

[34] ČMILANSKÁ, Silvia. *Škálovatelný dashboard* [online]. Pardubice, 2022 [cit. 2024-05-07]. Dostupné z: https://portal.upce.cz/StagPortletsJSR168/CleanUrl?urlid=prohlizeni-prace-detail&praceIdno=44073. Bakalářská práce. Univerzita Pardubice, Fakulta elektrotechniky a informatiky. Ing. Soňa Neradová, Ph. D.

[35] aaaanchakure. Introduction. *GeeksforGeeks: a computer science portal for geeks* [online]. Indie, Uttar Pradesh, Noida: GeeksforGeeks, 5. 4. 2024 [cit. 2024-05-07]. Dostupné z: https://www.geeksforgeeks.org/introduction-to-docker/.

[36] Networking overview. *Docker Documentation* [online]. USA, California: Docker Inc., 3. 5. 2024 [cit. 2024-05-07]. Dostupné z: https://docs.docker.com/network/.

[37] AWS. What is XML? - XML File Explained. *Cloud Computing Services: Amazon Web Services (AWS)* [online]. USA, Seattle, © 2024 [cit. 2024-05-07]. Dostupné z: https://aws.amazon.com/what-is/xml/.

[38] What is XML? Understand the Basics of Extensible Markup Language. *Marketing, Automation & Email Platform* [online]. USA, Georgie, © 2024 [cit. 2024-05-07]. Dostupné z: https://mailchimp.com/marketing-glossary/xml/.

[39] XML introduction - XML: Extensible Markup Language. *MDN Web Docs* [online]. USA, San Francisco, 10. 7. 2023 [cit. 2024-05-07]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction.

[40] XML Tree. *W3Schools Online Web Tutorials* [online]. Norway, Hommersåk, © 2024 [cit. 2024-05-07]. Dostupné z: https://www.w3schools.com/xml/xml_tree.asp.

[41] XML Namespaces. *W3Schools Online Web Tutorials* [online]. Norway, Hommersåk, © 2024 [cit. 2024-05-07]. Dostupné z: https://www.w3schools.com/xml/xml_namespaces.asp.

[42] XML, XLink and XPointer. *W3Schools Online Web Tutorials* [online]. Norway, Hommersåk, © 2024 [cit. 2024-05-07]. Dostupné z: https://www.w3schools.com/xml/xml_xlink.asp.

[43] XML and XPath. *W3Schools Online Web Tutorials* [online]. Norway, Hommersåk, © 2024 [cit. 2024-05-07]. Dostupné z: https://www.w3schools.com/xml/xml_xpath.asp.

[44] XPath Syntax. *W3Schools Online Web Tutorials* [online]. Norway, Hommersåk, © 2024 [cit. 2024-05-07]. Dostupné z: https://www.w3schools.com/xml/xpath_syntax.asp.

[45] XML Schema Tutorial. *W3Schools Online Web Tutorials* [online]. Norway, Hommersåk, © 2024 [cit. 2024-05-07]. Dostupné z: https://www.w3schools.com/xml/schema_intro.asp.

[46] BALAJI, Ramesh. Understanding XSD Schema. *CodeGuru: Microsoft Developer News* [online]. Spojené Arabské Emiráty, Dubaj, 10. 1. 2008 [cit. 2024-05-07]. Dostupné z: https://www.codeguru.com/csharp/understanding-xsd-schema/.

[47] XML Schema How To. *W3Schools Online Web Tutorials* [online]. Norway, Hommersåk, © 2024 [cit. 2024-05-07]. Dostupné z: https://www.w3schools.com/xml/schema_howto.asp.

[48] KŘEKOVÁ, Irena. 1. Kontroly výměnného formátu. *DTMwiki: Metodická pracovní skupina DTM* [online]. Česká Republika, Zlín: Krajský úřad Zlínského kraje. 25. 3. 2024 [cit. 2024-05-05]. Dostupné z: https://dtmwiki.kr-zlinsky.cz/kontroly/kontroly_jvf.

[49] KŘEKOVÁ, Irena. 1.4.2. Atributové kontroly. *DTMwiki: Metodická pracovní skupina DTM* [online]. Česká Republika, Zlín: Krajský úřad Zlínského kraje. 10. 4. 2024 [cit. 2024-05-05]. Dostupné z: https://dtmwiki.kr-zlinsky.cz/01_pravidla/04_kontroly/02_kontroly_atributy.

[50] KŘEKOVÁ, Irena. 1.4.3. Topologické kontroly. *DTMwiki: Metodická pracovní skupina DTM* [online]. Česká Republika, Zlín: Krajský úřad Zlínského kraje. 10. 4. 2024 [cit. 2024-05-05]. Dostupné z: https://dtmwiki.kr-zlinsky.cz/kontroly/kontroly_jvf.

[51] ČÚZK. ČÚZK - JVF DTM. *ČÚZK - Úvod* [online]. Česká Republika, Praha: ČÚZK. 7. 3. 2024 [cit. 2024-05-05]. Dostupné z: https://www.cuzk.cz/DMVS/JVF-DTM.aspx.

[52] ČÚZK. JVF DTM – Struktura formátu. *ČÚZK - Úvod* [online]. Verze 1.3. Česká Republika, Praha: ČÚZK. 31. 10. 2023 [cit. 2024-05-05]. Dostupné z: https://www.cuzk.cz/DMVS/JVF-DTM/JVF_DTM_143_StrukturaFormatu.aspx.

[53] ČÚZK. Report chyb při kontrole GAD v IS DTM krajů. *ČÚZK - Úvod* [online]. Verze 3.0. Česká Republika, Praha: ČÚZK. 13. 3. 2024 [cit. 2024-05-05]. Dostupné z: https://www.cuzk.cz/DMVS/Podklady-IS-DTM/Chybovy-XML-soubor/DTM_report_chyb_popis_v3.aspx.

[54] RabbitMQ. RabbitMQ: One broker to queue them all. *RabbitMQ: One broker to queue them all* [online]. UK, London: CloudAMQP, © 2024 [cit. 2024-05-07]. Dostupné z: https://www.rabbitmq.com/.

[55] JOHANSSON, Lovisa. Part 1: RabbitMQ for beginners - What is RabbitMQ? *CloudAMQP: Queue starts here.* [online]. Švédsko, Stockholm: CloudAMQP, 23. 9. 2019 [cit. 2024-05-07]. Dostupné z: https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html?gad_source=1&gclid=CjwKCAjw88yxBhBWEiwA7cm6pc7bec0Uc_cXxjIl0OT63mw69jAYrM1Awfe5Fhph53aeRFm7E3jCRBoCtioQAvD_BwE.

[56] PostgreSQL: About. *PostgreSQL: The world's most advanced open source database* [online]. USA, Philadelphia, © 2024 [cit. 2024-05-07]. Dostupné z: https://www.postgresql.org/about/.

[57] ZYLINSKI, Bartlomiej. What Keycloak Is and What It Does. *DZone: Programming & DevOps news, tutorials & tools* [online]. USA, Nashville, 11. 10. 2021 [cit. 2024-05-07]. Dostupné z: https://dzone.com/articles/what-is-keycloak-and-when-it-may-help-you.

[58] BAYOGLU, Mustafa Batuhan. What Is Keycloak? What Is Behind Of The Keycloak? How Does Keycloak Work?. *Medium* [online]. USA, San Francisco: Medium, 25. 6. 2023 [cit. 2024-05-07]. Dostupné z: https://bayoglubatuhan.medium.com/what-is-keycloak-what-is-behind-of-the-keycloak-how-does-keycloak-work-d1be3

[59] Microsoft. ASP.NET Core: Open-source web framework for .NET. *Microsoft – cloud, počítače, aplikace a hry* [online]. USA, Washington, © 2024 [cit. 2024-05-07]. Dostupné z: https://dotnet.microsoft.com/en-us/apps/aspnet.

[60] Overview of ASP.NET Core. *Microsoft – cloud, počítače, aplikace a hry* [online]. USA, Washington, 10. 4. 2023 [cit. 2024-05-07]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0.

[61] Simplilearn. Guide To Using Typescript With React. *Simplilearn: Online Courses - Bootcamp & Certification Platform* [online]. USA, San Francisco, 24. 2. 2023 [cit. 2024-05-07]. Dostupné z: https://www.simplilearn.com/tutorials/reactjs-tutorial/react-typescript.

SEZNAM PŘÍLOH

PŘÍLOHA A

Tato příloha obsahuje zdrojové kódy všech aplikací distribuovaného systému. Projekty jsou přiloženy v archivu s názvem CmilanskaS_DistribuovanySystem_SN_1cast_2024.zip.

PŘÍLOHA B

Tato příloha obsahuje Docker kontejnery a konfigurace. Data jsou přiložena v archivu s názvem CmilanskaS_DistribuovanySystem_SN_2cast_2024.zip.

PŘÍLOHA C

Tato příloha obsahuje testovací geodetická data ve formátu JVF DTM. Data jsou přiložena v archivu s názvem CmilanskaS_DistribuovanySystem_SN_3cast_2024.zip.