**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ**
**DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA**


# DISTRIBUOVANÝ A ROBUSTNÍ SYSTÉM PRO AUTOMATICKÝ PŘEPIS DOKUMENTŮ

**ROBUST AND DISTRIBUTED OCR PROCESSING SYSTEM**

**BAKALÁŘSKÁ PRÁCE**
**BACHELOR'S THESIS**


**AUTOR PRÁCE** **PAVEL RAUR**
**AUTHOR**

**VEDOUCÍ PRÁCE** **Ing. MICHAL HRADIŠ, Ph.D.**
**SUPERVISOR**


**BRNO 2021**

**Vysoké učení technické v Brně**
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2021/2022

# Zadání bakalářské práce

Student: **Raur Pavel**
Program: Informační technologie
Název: **Distribuovaný a robustní systém pro automatický přepis dokumentů**
**Robust and Distributed OCR Processing System**
Kategorie: Paralelní a distribuované výpočty

Zadání:

1. Seznamte se s přístupy pro tvorbu distribuovaných výpočetních aplikací, které umožňují řetězit moduly.
2. Seznamte se s nástroji pro komunikaci v distribuovaných aplikacích se zaměřením na robustní komunikaci přes sdílené fronty (např. Kafka, Zookeeper). Zároveň se seznamte s možnostmi vytvoření kontejnerů (například Docker).
3. Navrhněte distribuovaný systém pro OCR systém projektu PERO, který oddělí zpracování rozložení stránky dokumentu, přepis textu, dekódování textu pomocí jazykových modelů a který umožní řetězení těchto modulů.
4. Implementujte navržený systém ve spolupráci s týmem projektu PERO tak, aby řešení bylo možné integrovat do webové aplikace a API projektu.
5. Otestujte navržený systém v reálném provozu i syntetických testech.
6. Vytvořte krátké video shrnující vaši práci a její výsledky.

Literatura:

- Podle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Body zadání 1 až 3.

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

Vedoucí práce: **Hradiš Michal, Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 1. listopadu 2021

## Abstrakt

Tato práce se zabývá vytvořením distribuovaného systému pro přepis dokumentů za pomoci OCR. Navržený systém zahrnuje distribuci dat, koordinaci činnosti výpočetních uzlů a plánování zpracování. Dále se práce zabývá testováním vyvinutého systému. Vývoj je prováděn v rámci projektu PERO, kde bude výsledný software integrován do existující demonstrační aplikace.

## Abstract

This thesis focuses on creating distributed computing system for document processing using OCR. System is designed to coordinate computation across multiple nodes and distribute tasks between them. Created system was tested for proper functionality. Development is carried out within the PERO project. Developed system will be integrated into the project's demonstration application.

## Klíčová slova

distribuovaný systém, distribuce úloh, message broker, koordinace distribuovaného systému, OCR

## Keywords

distributed system, task distribution, message broker, distributed system coordination, OCR

## Citace

RAUR, Pavel. *Distribuovaný a robustní systém pro automatický přepis dokumentů.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Michal Hradiš, Ph.D.

# Distribuovaný a robustní systém pro automatický přepis dokumentů

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Michala Hradiše, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . . . . . . . . . . . . . . . . . . . . . .
Pavel Raur
6. května 2022

## Poděkování

Tímto bych rád poděkoval vedoucímu práce panu Ing. Michalu Hradišovi, Ph.D. za odborné vedení práce, ochotné rady a všechen čas strávený konzultacemi.

# Obsah


**1 Úvod** **2**

**2 Popis aktuální aplikace PERO a proč je potřeba nový výpočetní systém** **3**
2.1 Popis a parametry nového systému . . . . . 5

**3 Použité technologie a software** **7**
3.1 Distribuce úloh . . . . . 7
3.2 Koordinace a ovládání workerů . . . . . 10
3.3 Správa běhového prostředí . . . . . 10
3.4 Serializace úloh . . . . . 11

**4 Návrh systému pro přepis dokumentů** **12**
4.1 Návrh komponent systému . . . . . 12
4.2 Systém front . . . . . 15
4.3 Systém pro koordinaci a správu konfigurací . . . . . 15
4.4 Worker pro zpracování dokumentů . . . . . 16
4.5 Watchdog – Monitorování a automatická regulace . . . . . 17

**5 Implementace systému** **20**
5.1 Definice požadavku zpracování . . . . . 20
5.2 Systém front . . . . . 21
5.3 Systém pro koordinaci a správu konfigurací . . . . . 22
5.4 Implementační detaily workeru . . . . . 22
5.5 Implementační detaily watchdogu . . . . . 25
5.6 Ovládací skripty . . . . . 26

**6 Testování** **28**
6.1 Testování přepisu dokumentu . . . . . 30

**7 Závěr** **31**

**Literatura** **32**

# Kapitola 1

# Úvod

Tato práce vznikla v rámci projektu PERO [6]. Cílem projektu je zvýšení přístupnosti a využitelnosti digitalizovaných dokumentů. Za tímto účelem se projekt zabývá vývojem nástrojů pro automatizaci digitalizace dokumentů a zlepšení kvality digitalizovaných dokumentů. Využívá k tomu nástrojů jako počítačové vidění, strojové učení a jazykové modelování.

Cílem této práce je vytvoření systému, který zajistí rychlý přepis dokumentů, rozdělení přepisu do jednotlivých kroků, a řetězení těchto kroků. Součástí systému budou služby pro distribuci úloh výpočetním uzlům, správu konfigurací a monitorování těchto uzlů. Výpočetní uzly budou zajišťovat zpracování dokumentů, s využitím nástrojů vyvinutých v rámci projektu PERO. Celý systém poběží distribuovaně na více počítačích, kvůli rychlosti zpracování, ale i zajištění záloh jednotlivých částí systému v případě výpadku některého z počítačů.

Pro toto téma jsem se rozhodl kvůli zájmu o automatizaci a snaze o zjednodušení vyhledávání informací. Z mého pohledu může takováto aplikace pomoci s digitalizací starších záznamů, které jsou stále vedeny v archivech. Digitalizace může být často časově náročná a nákladná. Takovýto systém by pomohl s rychlou detekcí klíčových slov a kontextu, na jejichž základě by bylo možné dokument automaticky zařadit do digitální databáze a následně jej vyhledat. Díky tomu bude digitalizace dostupnější i tam, kde by se jí nikdo nezabýval z důvodů šetření financí, či času.

Tato práce je rozdělena do pěti kapitol. První z těchto kapitol obsahuje podrobnější informace o fungování aktuálního systému pro zpracování dokumentů, používaném v projektu PERO. Také jsou zde uvedeny důvody, proč je potřeba navrhnout nový systém. Následující kapitola shrnuje existující řešení a technologie, které se používají při implementaci obdobných systémů a lze je použít i v této práci. Následuje návrh systému pro přepis dokumentů, jeho rozdělení na jednotlivé podsystémy či komponenty a popis fungování těchto komponent. V předposlední kapitole se lze dočíst detaily implementace navrženého systému, a jaké technologie i dostupná softwarová řešení byly použity při implementaci. Poslední kapitola se zabývá testováním systému a jsou zde uvedeny některé postřehy a návrhy, co by bylo možné v návrhu systému zlepšit.

# Kapitola 2

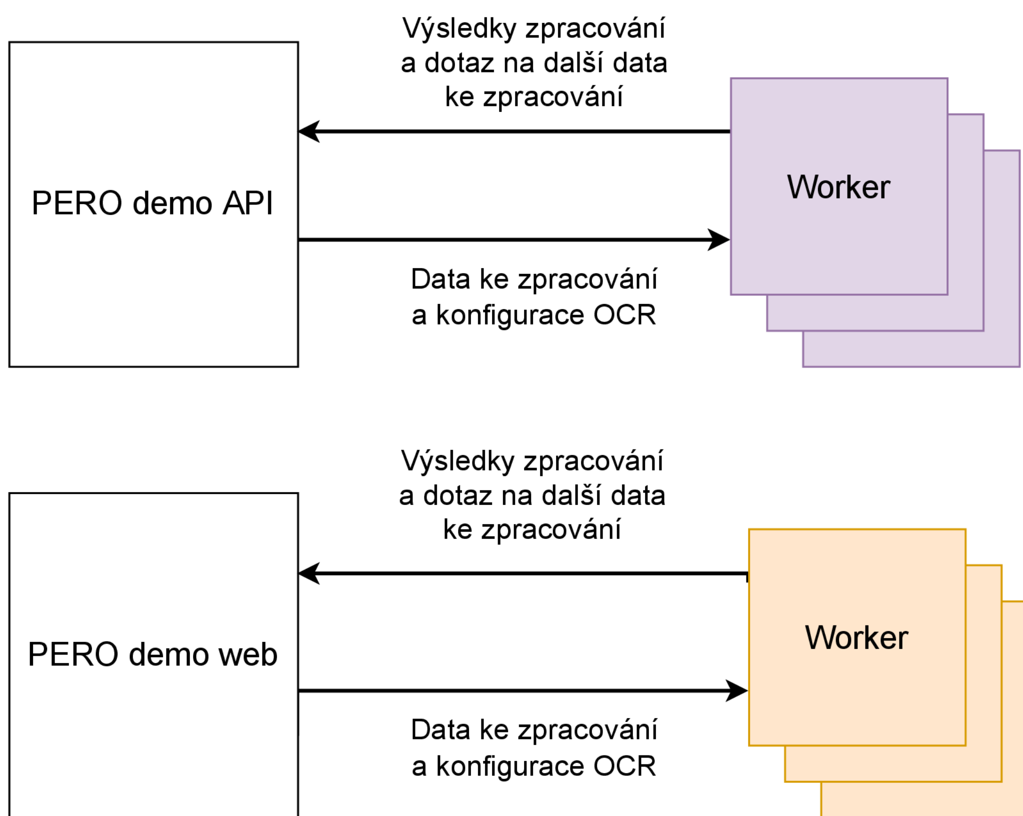
# Popis aktuální aplikace PERO a proč je potřeba nový výpočetní systém

Projekt PERO se zabývá vývojem *OCR* (*optical character recognition* – optické rozpoznání znaků) [17]. Tato technologie slouží k detekci textu v digitalizovaném dokumentu a jeho přepisu. Výsledkem je obvykle prostý text nebo jiný formát, který umožňuje další strojové zpracování dokumentu. Výsledky vývoje jsou dostupné v demonstrační aplikaci[1], kde si lze přepis dokumentů vyzkoušet.

Aplikace se aktuálně skládá z webového rozhraní, aplikačního rozhraní (API) a tzv. workeru. Do webového rozhraní či API mohou uživatelé nahrát dokumenty pro přepis, a po zpracování si je vyzvednout. Worker je zodpovědný za zpracování dokumentů pomocí OCR. Díky oddělení zpracování od uživatelského rozhraní, je možné provádět zpracování na více strojích najednou.

Worker komunikuje s aplikací pomocí HTTP. Periodicky se dotazuje, zda jsou dostupné dokumenty ke zpracování. Pokud nějaké dokumenty na zpracování čekají, stáhne je a lokálně zpracuje. Výsledky zpracování nahrává zpět do webové aplikace nebo API, dle konfigurace. Webová aplikace používá jinou skupinu workerů než API. Nastavení si worker stahuje společně s daty ke zpracování. Zpracování pak probíhá různým způsobem. Dokumenty z API jsou zpracovávány po stránkách, zatímco dokumenty z webového rozhraní jako celek. Po zpracování dokumentu se worker restartuje a znovu se dotazuje, zda nějaké dokumenty čekají na zpracování. V případě chyby workeru je zde nutný manuální zásah administrátora, který daný problém vyřeší. Diagram komunikace komponent v aplikaci znázorňuje obrázek 2.1.

[1] Demonstrační aplikace projektu PERO: `https://pero-ocr.fit.vutbr.cz/`.




Obrázek 2.1: Schéma aktuálního systému. Skupiny workerů se připojují ke každé aplikaci zvlášť. Workery tedy nelze jednoduše sdílet.

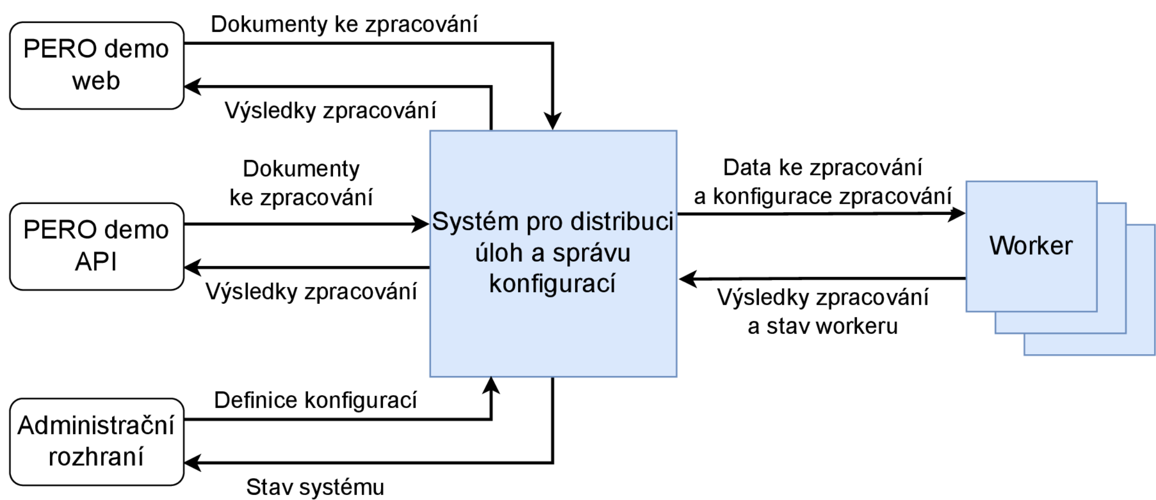
## 2.1 Popis a parametry nového systému

Nový systém, jehož vývojem se tato práce zabývá, je určen k rozšíření a částečnému nahrazení aktuální demonstrační aplikace popsané v kapitole 2. Zajistí lepší správu workerů a umožní lepší administraci a škálování. Zde je soupis požadavků, které má nový systém splňovat:

- *Robustnost* – zahrnuje obnovu z chyb. Případná chyba by neměla způsobit selhání celého systému. Při závažném problému je upozorněn administrátor.
- *Distribuovanost* – týká se nejen zpracování dokumentů, které musí probíhat na více počítačích současně, ale i robustnosti. Zahrnuje replikaci dat, takže při selhání některé části systému jsou dokumenty, nahrané do systému, stále dostupné z jiného zdroje.
- *Rozdělení zpracování na mezikroky* – jednotlivé kroky zpracování budou probíhat samostatně, namísto zpracování celého dokumentu naráz, které je prováděno nyní. Zároveň se sjednotí zpracování pro webovou aplikaci i API a zjednoduší se konfigurace OCR, která bude uložena na jednom místě, a bude sdílená pro webové rozhraní i pro API.
- *Správa konfigurací pro zpracování dat* – každému kroku zpracování je přidělena samostatná konfigurace pro OCR.
- *Automatická regulace na základě aktuálního stavu systému* – systém bude zajišťovat automatické monitorování zpracování a upravovat činnost workerů tak, aby předcházel zahlcení a dokumenty byly zpracovány co nejrychleji.

Jedním z požadavků je také vytvoření mezivrstvy mezi workery a webovým rozhraním či API. Tato mezivrstva slouží jako zprostředkovatel zpráv mezi jednotlivými částmi systému. Přebírá zodpovědnost za doručení požadavků zpracování workerům a výsledků zpět zadávající aplikaci. Díky této vrstvě je možné do stejného systému nahrávat požadavky z více různých uživatelských aplikací a sdílet workery mezi nimi mnohem jednodušeji, než když se workery připojují přímo na zadávající aplikace. Tímto se zlepší škálovatelnost a rozšiřitelnost celé aplikace PERO. Schéma nového systému zobrazuje obrázek 2.2. Zároveň jsou zde uvedeny nároky, které jsou na tuto propojující vrstvu kladeny. Tyto nároky vyplývají z předchozích požadavků na parametry systému a jsou následující:

- *Spolehlivé doručení požadavků* – doručení, ale i uložení požadavků na zpracování dokumentů musí být spolehlivé. Požadavky musí „přežít“ restart počítače, a to i neočekávaný (výpadek proudu, atd.), až do chvíle, než jsou zpracovány. Totéž platí pro výsledky zpracování.
- *Priority požadavků* – požadavky mohou mít různou prioritu. Ty s větší prioritou budou doručeny přednostně.
- *Vysoká dostupnost* (anglicky *high availability*) – software realizující tuto mezivrstvu musí běžet distribuovaně, aby i při výpadku byla data dostupná z jiného zdroje [10].




Obrázek 2.2: Základní schéma nového systému na zpracování požadavků. Je zde patrné oddělení výpočetní části aplikace (modře) od uživatelských rozhraní či aplikací. Systém nyní umožňuje sdílení konfigurací a workerů pro zpracování mezi jednotlivými aplikacemi.

# Kapitola 3

# Použité technologie a software

Zde je vypracována analýza existujících technologií, které řeší problémy probrané v kapitole 2. Zároveň ale nejde o přehled všech dostupných technologií, které mohly být použity pro implementaci. Jde spíše o stručný popis a uvedení čtenáře do kontextu. Jsou zde uvedeny a stručně popsány možnosti dostupného software vhodného pro použití v této práci, případně i alternativ řešících podobné problémy. V kapitole 5 je pak uvedeno, které z těchto nástrojů a softwaru byly dále použity a proč.

Na úvod je také dobré říci, že jako implementační jazyk byl zvolen python3. To vyplývá z požadavků projektu PERO, jehož OCR knihovny jsou taktéž implementovány v tomto jazyce. Bylo by nepraktické používat je z jiného jazyka a muset řešit z toho vyplývající problémy s kompatibilitou. Navíc je tento jazyk pro implementaci vhodný i díky velkému množství dalších knihoven, které mohou pomoci zrychlit vývoj, jak je patrné z následujících informací v této kapitole. Díky těmto skutečnostem zde o použití jiného jazyka nebylo uvažováno.

## 3.1 Distribuce úloh

Jak je uvedeno v kapitole 2.1, jeden z řešených problémů je distribuce úloh mezi jednotlivými částmi systému. K tomuto účelu existuje specifický typ programu, který se nazývá *message broker* (*zprostředkovatel zpráv*). Message broker zprostředkovává přenos *zprávy* (*message*), v tomto případě úlohy, mezi zdrojovou a cílovou službou. Zprávy jsou obvykle zařazovány do *fronty* (*queue*), která funguje na principu FIFO (first in, first out – první dovnitř, první ven) [3]. Je zde tedy možné zajistit zpracování v pořadí, v jakém byly úlohy zadány. Při hodnocení dostupných message brokerů pro případné použití v této práci, byly kromě parametrů, které musí nový systém splňovat, brány v úvahu i další požadavky. Mezi ně patří dostupnost knihoven pro použití tohoto brokeru z programovacího jazyka python3 a možnost jednoduchého monitorování stavu front. V neposlední řadě byl brán ohled také na kvalitu a přehlednost dokumentace.

### Apache ActiveMQ

Prvním porovnávaným brokerem je Apache ActiveMQ [1]. Tento message broker poskytuje rychlý přenos zpráv a podporuje protokol JMS (Java message service), který je specifický pro jazyk Java. Podporuje ale i další protokoly jako *AMQP* (advanced message queuing protocol – dalo by se přeložit jako protokol pokročilého řazení zpráv), který je dobrým standardem mezi message brokery [3]. Dle dokumentace podporuje persistenci zpráv ve

frontách, distribuci zpráv z froty více *konzumentům* (*consumer* – aplikace zpracovávající zprávy z message brokeru), zapojení broker serverů do clusteru a další možnosti [1].

Také existuje verze Apache ActiveMQ Artemis [2], která má být novou generací ActiveMQ. Tato verze má být rychlejší a poskytovat další funkce. Aktuálně tato verze ActiveMQ není schopna nahradit „klasickou“ verzi, která je v současnosti stále rychlejší a má větší propustnost zpráv [3]. Tudíž o použití této verze nebylo uvažováno.

Nedostatky ActiveMQ jsou především dokumentace, kde je, alespoň v porovnání s dokumentacemi ostatních brokerů, občas problém najít některé možnosti nastavení. Také chybí detailnější popis komunikace s klienty a detaily rebalancování zpráv ve frontě.

### Apache Kafka

Message broker Apache Kafka [22] je specifický svojí specializací na vysokou propustnost zpráv. Jak je popsáno na jeho webových stránkách a v dokumentaci, jde především o platformu pro rychlé „streamování událostí“ [3]. Kafka kromě vysoké propustnosti poskytuje replikování front. Předpokládá běh v *cluster*u, kde je více serverů propojeno do distribuovaného systému, který funguje jako jeden message broker. Tímto je zároveň zajištěna vyšší robustnost a tolerance chyb – při výpadku serveru jsou fronty dostupné na jiném serveru v clusteru [22].

K zajištění rychlého průchodu zpráv systémem je každá zpráva zapsána do tématu, pod kterým jsou registrováni konzumenti, kteří chtějí přijímat zprávy z tohoto tématu. Každý konzument má svoji frontu, která je implementována jako log zpráv. Nová zpráva je přidána na konec tohoto logu. Výhodou tohoto brokeru je možnost konzumenta pohybovat se ve frontě zpět do historie a znovu zpracovávat již zpracované zprávy. Nevýhodou může být to, že celá fronta je uložena na serveru, protože je implementována jako log. Dalším problémem může být rebalancování zpráv. Zde může dojít k opakovanému zpracování již zpracovaných zpráv, nebo vynechání některých zpráv, které nebyly zpracovány vůbec. Záleží na konfiguraci pro dané téma [22].

Tento message broker má velmi dobrou dokumentaci, kde lze najít detailní popis fungování brokeru a jeho chování v různých situacích. Je zde také popis nastavení, různé tutoriály a demonstrační videa.

Existuje několik knihoven pro práci s tímto brokerem z jazyka python3. Lze jmenovat například knihovnu *confluent_kafka* od Confluent Inc., která poskytuje kompletní podporu brokeru a velké množství funkcí [5]. Alternativně je použitelná i kompaktnější knihovna *kafka-python*, která se naopak snaží o jednoduché použití brokeru [13].

### RabbitMQ

RabbitMQ [24] je velmi rozšířený message broker s dobrými vlastnostmi, co se týče propustnosti a rychlosti přeposílání zpráv [3]. Jeho hlavní podporovaný protokol je již zmíněný *AMQP* ve verzi *0–9–1*, který poskytuje mnoho vlastností. RabbitMQ navíc implementuje i mnoho rozšíření tohoto protokolu. Dále podporuje protokoly STOMP, MQTT a další. Z podporovaných vlastností stojí za zmínění perzistence zpráv ve frontách, podpora priorit zpráv, podpora replikace front napříč více servery [3]. Z popisu fungování je velmi podobný ActiveMQ, zmíněnému výše v této kapitole. Oproti ActiveMQ dosahuje RabbitMQ lepších výsledků co se týče propustnosti při velkém objemu přeposílaných dat [3].

Velkým plus je kvalitní dokumentace, kde je detailně popsáno fungování brokeru. Také jsou dostupné tutoriály pro práci s brokerem z různých programovacích jazyků, včetně pythonu3. Na rozdíl od brokeru Apache Kafka, tento broker umožňuje vytvoření jediné fronty,

která je sdílena mezi více konzumenty. Fronta má hlavní, sdílenou část, která umožňuje použití priorit. Broker rozděluje zprávy z této hlavní sdílené fronty rovnoměrně mezi připojené konzumenty. Každý konzument má pak svoji vlastní krátkou frontu. V této frontě jsou zprávy čekající na zpracování tímto konkrétním konzumentem. Délku této krátké fronty lze upravit parametrem `prefetch count`, který si konzument nastavuje dle potřeby. Tato krátká fronta je navíc duplikována na brokeru i na konzumentovi, což umožňuje rychlejší zpracování, protože konzument nečeká na stažení zpráv z brokera. Zároveň jsou tyto zprávy stále dostupné na brokerovi, takže při odpojení konzumenta je možné jeho nezpracované zprávy přemístit zpět na začátek sdílené hlavní fronty, odkud budou následně předány ke zpracování jiným konzumentům.

Je tedy zřejmé, že na rozdíl od Apache Kafka, kde je fronta spíše log zpráv [22], v RabbitMQ jde více o tradiční pojetí fronty. Zprávy jsou po zkonzumování z fronty odebrány a není možné se k nim vracet. Pro zajištění spolehlivého zpracování broker čeká na potvrzení, že zpráva byla úspěšně zkonzumována. Až poté je odebrána z fronty [24].

Komunikace a práce s brokerem je možná z velkého množství programovacích jazyků. Na stránkách RabbitMQ je kompletní výpis podporovaných jazyků i knihoven. Z pythonu je možné se k tomuto brokeru připojit pomocí knihovny *pika* [7]. Tato knihovna umožňuje komunikaci pomocí AMQP protokolu. Také umožňuje deklaraci front na brokeru a nastavení parametrů spojení. Navíc je v oficiální dokumentaci RabbitMQ tutorial, jak s brokerem pracovat za použití právě této knihovny [24].

### Apache Qpid

Apache Qpid je message broker podobný ActiveMQ, umožňuje komunikaci pomocí AMQP protokolu. Existují dvě verze tohoto brokeru, které jsou implementovány v jazycích *C++* a *Java* [21]. Broker poskytuje podobné možnosti jako ActiveMQ, nicméně nedosahuje možností RabbitMQ. Ani dokumentace není tolik obsáhlá jako u zmíněných dvou brokerů. Kvůli tomuto nedostatku broker nebyl dále uvažován pro použití v tomto projektu.

### ZeroMQ a Celery

Dále byly porovnány samostatné knihovny, o jejichž použití bylo uvažováno při implementaci workeru. Tyto knihovny jsou ZeroMQ a Celery. Obě jsou samostatné knihovny, nejde tedy o message broker.

ZeroMQ je knihovna k implementaci celkového řešení front v projektu [23] – s její pomocí lze naimplementovat komunikaci pomocí front, vlastní message broker, atd. Nejde tedy přímo o podpůrné funkce pro napojení na konkrétní message broker, nebo pro použití konkrétního již existujícího protokolu. Jedná se sice o zajímavou možnost, ale pro tento projekt není vhodná právě proto, že je zde potřeba implementovat celé vlastní řešení komunikace. Jednalo by se o zbytečnou práci navíc, a je vhodnější použít existující řešení.

Celery je další knihovna, která byla zvažována pro implementaci. V tomto případě se jedná o univerzální knihovnu pro klienta (konzumenta) v jazyce python3. Umožňuje připojení k brokerům používajícím AMQP protokol – doporučovaný je broker RabbitMQ. Také umožňuje použití cache a dalších funkcí [16].

## 3.2 Koordinace a ovládání workerů

Další z problémů, které systém řeší, je jednoduché ovládání workerů, ideálně bez nutnosti připojovat se na počítač, kde worker běží, a měnit zde jeho konfiguraci. Zároveň je třeba zajistit synchronizaci systému a umožnit změnu nastavení podle aktuálního stavu. Tyto problémy může řešit software pro *koordinaci distribuovaného systému*, který umožňuje centrální řízení a reportování stavu jednotlivých částí systému [20]. Díky tomu je možné takovýto software použít pro uložení konfigurací, ale i pro zasílání řídících signálů mezi jednotlivými službami.

V této sekci dominuje zejména *Apache Zookeeper* [20]. Tato služba umožňuje zmíněnou koordinaci systému. Poskytuje API podobné struktuře souborového systému, kde lze vytvářet *uzly*, (*node* nebo *znode*). Ty jsou použity k uložení informace, podobně jako soubory. Případně mohou mít poduzly, což je analogické s adresářem v souborovém systému. Rozdíl je v tom, že uzel může zároveň mít poduzly, i obsahovat data. Data uložená v uzlu mohou dosahovat maximální velikosti několika kilobytů, tudíž nejde přímo o klasické soubory, určené pro uložení velkých objemů informací. API Zookeeperu umožňuje definovat tzv. *watch*. Jde o registrovanou akci, která při změně uzlu automaticky upozorní klienta, který si watch nadefinoval, že ke změně došlo. Připojené služby tudíž nemusí neustále kontrolovat stav systému, konfigurací a dalších uložených informací. Při změně jim je aktualizace zaslána automaticky. Zookeeper dokáže běžet distribuovaně a stav serverů je synchronizován. Díky tomu je v případě selhání některého serveru k dispozici záloha. Operace v Zookeeperu poskytují garance jako je atomicita operací, proběhnutí operací v zadaném pořadí, zajištění aktuálních dat napříč servery v clusteru a další [19].

Alternativ k funkci Zookeeperu není mnoho. Existuje *Chubby*, což je podobný software, který slouží především pro poskytování distribuovaných zámků. Jeho vývojářem je Google, který jej vyvinul pro interní použití. Google také vydal dokument popisující, jak tento software funguje [4]. Nicméně Chubby není volně dostupný a nelze jej tedy použít v této práci.

## 3.3 Správa běhového prostředí

Správou běhového prostředí se zde myslí izolace a zapouzdření programů a služeb. Toto umožňuje jednodušší správu aplikace a knihoven, které potřebuje ke svému běhu. Také je možné běžící procesy izolovat od procesů jiných aplikací [11]. V dnešní době je v této oblasti dostupná řada alternativ. Zde byly vybrány dvě možnosti, které jsou velmi často používané, a to *Docker* a *Kubernetes*.

Docker je platforma pro tvorbu a spouštění *kontejnerů* (*container* – zapouzdřené aplikační prostředí) [11]. Kontejnery jsou většinou založené na nějaké linuxové distribuci a poskytují minimální běhové prostředí, nutné pro běh základních programů operačního systému. Do tohoto prostředí je následně nainstalována aplikace spolu s jejími závislostmi (knihovny, interpreter jazyka, atd.). Aplikace má přístup pouze ke zdrojům dostupným v kontejneru. Docker umožňuje správu zdrojů kontejneru – omezení výpočetního výkonu, přístupu k paměti, disku, atd. Zároveň umožňuje úpravu konfigurací aplikace pomocí konfiguračních proměnných, otevření portů pro síťový provoz, a další funkcí. Díky tomu je možné stejnou aplikaci spustit několikrát s různou konfigurací, případně i více různých verzí aplikace, bez toho, aby o sobě tyto instance „věděly“ a mohly se ovlivňovat. Docker

taktéž poskytuje registr kontejnerů *Docker Hub*[1]. Odtud je možné stáhnout předpřipravené kontejnery, například pro většinu message brokerů, zmíněných v kapitole 3.1.

Existují další rozšíření této platformy, poskytující takzvanou *orchestraci* (*orchestration*) kontejnerů. Orchestrace zahrnuje automatickou konfiguraci, správu a koordinaci kontejnerů [14]. Mezi často používané patří například zmíněný Kubernetes [18]. Tato platforma běží distribuovaně na více počítačích, nad kterými vytváří abstrakci. Tímto je možné spravovat kontejnery nezávisle na tom, který počítač je hostuje. Dále Kubernetes umožňuje automatické *vyvažování zátěže* (*load balancing*). Automaticky umí spouštět a zastavovat kontejnery podle vytížení aplikace, nebo při selhání kontejneru, či některého z počítačů v clusteru. Za zmínku stojí možnost správy konfigurace a možnost spouštění *dávkových úloh* (*batch execution*). Díky tomu je možné v kontejneru spustit úlohu, která neběží jako služba, ale ukončí se po vypočítání výsledku.

## 3.4 Serializace úloh

Během návrhu systému byl jeden z řešených problémů rychlost přenosu, serializace a deserializace dat. K řešení bylo porovnáno několik technologií. Kromě tradičního *JSON* (Javascript Object Notation – Javascriptový zápis objektu), byly porovnány dva binární formáty dat a jejich API.

*Protobuf* [8] (*Protocol Buffers*) je první z těchto formátů. Jde formát, který byl vyvinut firmou Google. Jde o binární formát pro serializaci dat, kde schéma formátu je předem definované. Nelze jej tedy libovolně rozšířit po vytvoření, jako například JSON. Formát zprávy je definován v jazyce Protobufu a následně je pomocí proto-compileru vygenerován kód pro práci s tímto formátem ve specifikovaném programovacím jazyce. Díky tomu je formát kompatibilní napříč programovacími jazyky a zároveň nezávislý na nějakém konkrétním. Vygenerovaný kód umožňuje práci s přijatou zprávou, která je reprezentována objektem daného programovacího jazyka. Jednotlivá pole zprávy jsou atributy tohoto objektu. Datové typy jsou taktéž přeloženy, takže lze s daty zprávy jednoduše pracovat.

*Flatbuffers* [9] je druhý z porovnávaných formátů. Funguje podobně jako Protobuf, kterým je inspirovaný. V podstatě jde o novější verzi téhož formátu. Rozšíření, která Flatbuffers přináší, se týkají rychlosti změn dat ve zprávě. Zatím co Protobuf je vždy deserializován do objektu, Flatbuffers umožňuje tzv. *Mutaci* (*Mutation*), což je změna hodnoty proměnné přímo v serializovaném objektu. Díky tomu se přijatá zpráva nemusí deserializovat, pokud to není nutné. Výhody jsou nižší paměťová náročnost a rychlejší zpracování. Tato vlastnost Flatbufferu není zatím bohužel dostupná v jazyce python3, protože ji API pro tento jazyk nepodporuje. Kvůli tomu jsou Flatbuffers i Protobuf v pythonu3 téměř ekvivalentní.

[1]Registr kontejnerů Docker Hub: `https://hub.docker.com/`.

# Kapitola 4

# Návrh systému pro přepis dokumentů

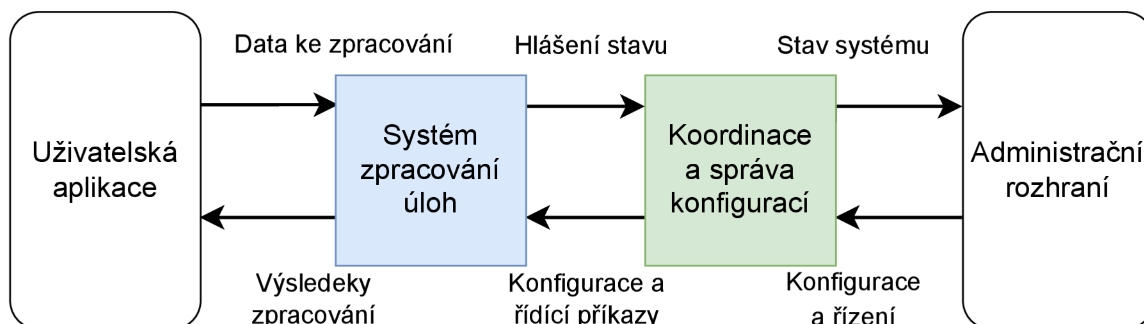
Tato kapitola se zabývá popisem návrhu systému pro přepis dokumentů. Jsou zde popsány jednotlivé části systému, jaké jsou jejich úkoly, a jak spolu interagují.

Jednotkou zpracování dat v systému je stránka dokumentu. Ta je zpracovávána v *etapách* (*stage*), které dohromady tvoří tzv. *pipeline* – posloupnost kroků zpracování. Každá stage má svoji konfiguraci, která definuje, jakým způsobem bude stránka v tomto kroku zpracována. Jednotlivé stage očekávají různé vstupy – může jít jen o samotnou stránku dokumentu, nebo i nějaká metadata, vygenerovaná v předchozí stage. Konfigurace jsou spravovány systémem. Je možné je definovat při vytvoření nové stage, nebo je dodatečně upravit. Stage lze libovolně přidávat a odebírat. Uživatel definuje pipeline z existujících stage, které může libovolně zřetězit za sebe, aby dosáhl požadovaného výsledku. Systém nezná kontext stage a nevaliduje pipeline. Je na uživateli, či na zadávající aplikaci, aby vytvořila pipeline, která vrátí smysluplný výsledek zpracování.

## 4.1 Návrh komponent systému

Jak bylo popsáno v kapitole 2.1 a je znázorněno na obrázku 2.2, nový systém bude sloužit jako mezivrstva mezi uživatelským rozhraním a workery, které zpracovávají data. Bude zajišťovat distribuci úloh mezi workery, správu konfigurací workerů a automaticky regulovat jejich činnost na základě stavu systému. V kapitole 3 byly popsány technologie, které lze použít k řešení těchto požadavků. Z analýzy těchto technologií vyplývá, že některé z požadovaných funkcí může provádět již dostupný software. Nicméně není jednoduše možné přidat jedinou monolitickou komponentu, která by zajišťovala všechny zmíněné funkce. Nebylo by to ani praktické, z důvodů údržby a dalšího vývoje systému. Proto je zde tento systém rozdělen na komponenty, o kterých lze uvažovat jako o *mikroslužbách* (*microservice*) [15][11]. Každá komponenta systému řeší specifickou část problému a dohromady vytvářejí řešení. Výhodou je, že tyto oddělené služby je možné jednodušeji spravovat, rozšiřovat, případně nahradit. Díky tomu je systém modulární a může být provozován distribuovaně – každá část poběží na jiném počítači. Zároveň je zde řešena robustnost, protože komponenty, řešené jako mikroslužby, mohou běžet paralelně a mohou být redundantní [11].

Z logického pohledu se celý systém dělí na dva hlavní podsystémy. Ty slouží spíše jako abstrakce, než že by každou z nich implementovala jedna aplikace či služba. Tyto podsystémy jsou *systém zpracování úloh* a *systém pro koordinaci a správu konfigurací*. Toto




Obrázek 4.1: Logické dělení systému na podsystémy.

dělení ilustruje obrázek 4.1. Toto dělení vychází z faktu, že systém musí zajišťovat dvě hlavní funkce. Jedna je přenos a zpracování variabilních dat (dokumentů), která do systému přichází a po zpracování z něj odchází. Druhá z těchto funkcí je správa stavu systému, který zahrnuje konfigurace a stav připojených workerů. Tyto informace jsou v systému uloženy a neopouštějí jej. Namísto toho se v čase mění. Do správy systému lze zahrnout i předávání řídících příkazů, které způsobují změnu stavu, ale i monitorování, které je možné taktéž považovat za součást řízení.

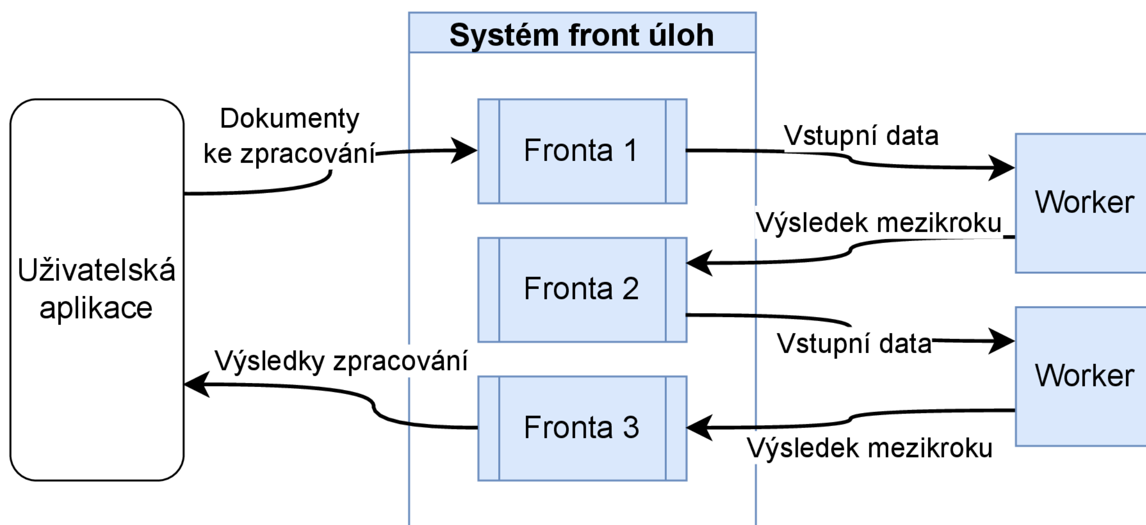
Každý z těchto podsystémů se dělí na komponenty, které jsou již konkrétními mikroslužbami. Pro realizaci je použito dostupné řešení, nebo je potřebný software implementován v rámci této práce. Komponenty podsystému pro zpracování úloh jsou tyto:

- Služba realizující systém front pro distribuci úloh
- Worker pro zpracování úloh

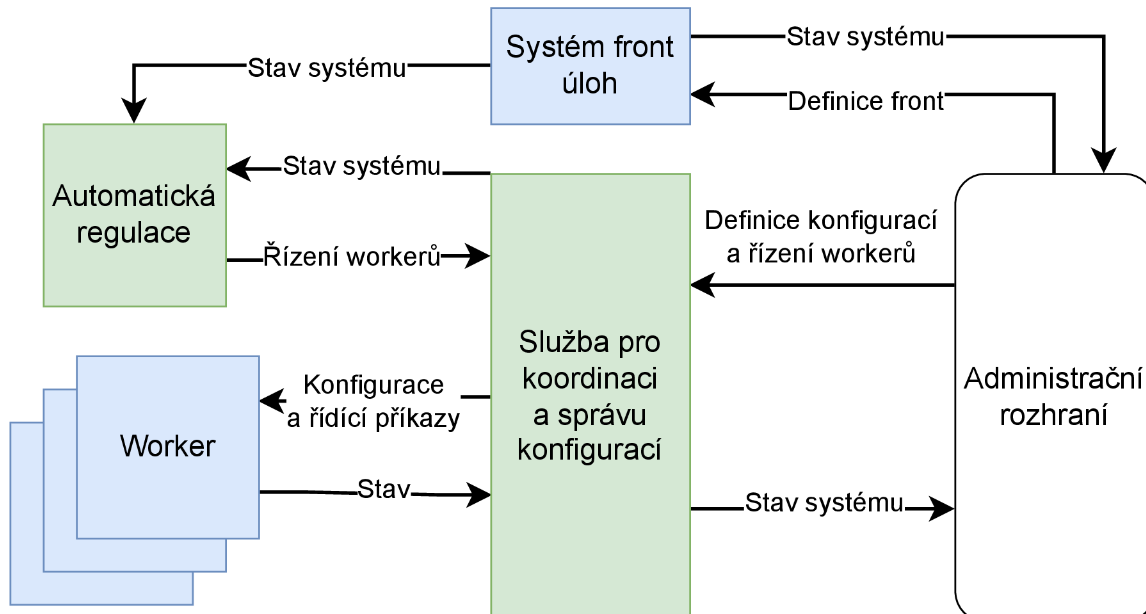
Komponenty podsystému pro koordinaci a správu konfigurací jsou tyto:

- Služba pro koordinaci, správu konfigurací a řízení workerů
- Watchdog – služba pro automatickou regulaci zpracování na základě aktuálního stavu

Jak bylo zmíněno v kapitole 2, v projektu bude použita existující webová stránka demonstrační aplikace projektu PERO a existující API. V obrázku 4.1 je také zmíněno administrační rozhraní. I když implementace těchto rozhraní není primární náplní projektu, bude třeba vyvinout i několik nástrojů pro ovládání systému. Hlavním důvodem je, aby bylo možné tento systém zprovoznit, testovat a monitorovat. Následující dva obrázky ilustrují rozdělení systému na podsystémy a znázornění interakce komponent v podsystémech. Na obrázku 4.2 je patrný tok dat podsystémem pro zpracování úloh. Obrázek 4.3 znázorňuje předávání konfigurací, monitorovacích informací a řídících příkazů mezi různými částmi systému.




Obrázek 4.2: Schéma podsystému pro zpracování úloh a zachycení toku dat systémem. Uživatelskou aplikací je zde myšleno webové rozhraní či API demonstrační aplikace PERO. Aplikací může být i více.




Obrázek 4.3: Schéma předávání řídících příkazů a konfigurací v systému. Komponenty podsystému pro zpracování úloh jsou modře, komponenty pro správu konfigurací a koordinaci systému jsou vyznaženy zeleně.

## 4.2 Systém front

Systém front slouží pro distribuci úloh workerům. Z analýzy dostupného software vyplývá, že tuto komponentu bude realizovat message broker. Popisy dostupných brokerů jsou k dispozici v kapitole 3.1. Message broker by měl splňovat následující požadavky:

- *Perzistence zpráv ve frontách* – message broker musí zajistit, že při výpadku napájení nebudou úlohy čekající na zpracování ztraceny.
- *Priority zpráv* – broker by měl umožňovat definici priority úlohy; úlohy s vyžší prioritou budou předány konzumentům přednostně.
- *Replikace front* – fronty zpráv mohou být replikovány na více serverech, takže při výpadku jednoho serveru jsou stále dostupné z jiného běžícího serveru.
- *Spolehlivé doručení zpráv* – broker neodstraní zprávu z fronty, dokud nedostane potvrzení, že byla úspěšně zkonzumována.
- *Spolehlivé přijetí zpráv* – broker umožňuje odeslání potvrzení zadavateli zprávy, že zpráva byla zařazena do fronty.
- *Podpora pro python3* – jsou dostupné knihovny pro práci s tímto brokerem, respektive s protokolem, kterým komunikuje, z jazyka python3.

## 4.3 Systém pro koordinaci a správu konfigurací

Jak název kapitoly napovídá, pro správu konfigurací a ovládání workerů byl z dostupného software vybrán software pro koordinaci distribuovaného systému, detailněji popsaný zde 3.2.

U této části systému byly zvažovány dvě možnosti. První byla správa workerů pomocí jejich běhového prostředí. Tedy technologie pro orchestraci kontejnerů, kde externí služba spustí worker v kontejneru a nakonfiguruje ho. Kontejnery s workery jsou na základě aktuálních požadavků spouštěny s různými konfiguracemi. Při změně konfigurace je třeba všechny běžící kontejnery vypnout a spustit znovu s novou konfigurací. Možnosti orchestrace kontejnerů jsou probrány v kapitole 3.3. Druhou možností je správa konfigurací pomocí aplikace pro koordinaci distribuovaného systému, které jsou uvedeny v kapitole 3.2. Zde se worker registruje do služby, která provádí koordinaci činnosti a správu nastavení. Worker tedy není vypínán a spouštěn znovu s novou konfigurací, ale tato koordinační služba jej upozorní, pokud se změní nastavení a worker na toto upozornění musí sám reagovat. Zároveň worker může reportovat svůj stav a statistiky do této služby. Interakce tedy může probíhat oběma směry.

Pro realizaci této části systému byl zvolen software pro koordinaci distribuovaného systému, protože umožňuje koordinovat činnost workerů bez ohledu na jejich běhové prostředí. Pokud by byl zvolen kontejnerový orchestrátor, workery by musely běžet v kontejnerech, aby je tento nástroj mohl spravovat. To znamená, že je třeba orchestrátor zprovoznit na všech serverech, kde bude docházet ke zpracování dat. Aktuálně není na serverech, kde budou běžet workery, žádný takový nástroj dostupný. OCR, použité pro zpracování dat, navíc může potřebovat přístup ke grafické kartě kvůli akceleraci výpočtu. Bylo by třeba zajistit, že grafická karta může být použita z kontejneru, který ji právě potřebuje. Z tohoto důvodu může být nastavení jakékoliv izolace běhového prostředí náročné či nemožné. Použití systému pro

koordinaci, který interaguje s workerem pomocí standardní síťové komunikace, umožňuje worker nainstalovat do prostředí operačního systému, který na serveru běží. Tato možnost odstraňuje potenciální problémy se správou konfigurace a řízením workeru v případě, že se nepodaří zprovoznit kontejnery a nastavit izolaci běhového prostředí. Navíc použití kontejnerů není vyloučeno, a pokud to bude možné, může worker běžet v kontejneru. V tomto případě jde ale jen o možnost, nikoliv nutnost.

Systém pro koordinaci činnosti workerů by měl splňovat tyto požadavky:

- *Redundance* – služba musí běžet distribuovaně na více serverech, takže při výpadku je stále dostupný z jiného serveru.
- *Uložení konfigurací a dat* – služba zajišťuje uložení konfigurací a statistik v textové i binární podobě.
- *Notifikace při změně* – při změně uložených dat je služba schopna upozornit připojené služby, že tato změna proběhla.
- *Synchronizace* – služba poskytuje nástroje pro synchronizaci paralelně běžících procesů a řešení *souběhu* (*race condition*) při přístupu k datům.
- *Podpora pro python3* – existují knihovny pro práci s touto službou z pythonu3.

## 4.4 Worker pro zpracování dokumentů

Worker, který je aktuálně součástí aplikace, nevyhovuje požadavkům nového systému. Proto bude potřeba naimplementovat nový. Aktuální worker je popsaný v kapitole 2.

Pro implementaci nového workeru byl, stejně jako u starého workeru, zvolen jazyk python3. Hlavním důvodem je, že knihovny pro práci s OCR, vyvinuté v projektu PERO, jsou taktéž naprogramovány v tomto jazyce, takže je to nejjednodušší možnost. Funkce a parametry nového workeru jsou následující:

- *Podpora message brokeru* – worker bude přijímat úlohy a odesílat výsledky přes systém front, který bude realizovat message broker.
- *Podpora koordinace a vzdálené konfigurace* – worker bude reagovat na změnu konfigurace, která mu bude předána ze systému pro koordinaci.
- *Hlášení stavu* – worker bude svůj stav reportovat do systému pro koordinaci
- *Potvrzení zpracování úlohy* – worker zašle potvrzení o zpracování úlohy message brokeru, který až po obdržení potvrzení odebere úlohu z fronty.
- *Loggování činnosti* – worker generuje log prováděných akcí během dané stage, tento log je zahrnut mezi výslednými daty kvůli případnému debugování zpracování v dané stage. Také generuje log všech akcí, které provádí, aby bylo možné dohledat i případné problémy, které se týkají workeru samotného, nejen zpracování a OCR.

Práce workeru bude probíhat následujícím způsobem:

1. Registrace workeru v systému pro koordinaci
2. Stažení nastavení
   inicializace nastavení workeru

3. Připojení k systému front
4. Zpracování zadaných úloh dokud není obdržen signál k vypnutí
5. Deinicializace workeru a ukončení

V bodě 4. bude worker zpracovávat úlohy, dokud budou nějaké k dispozici ve frontě pro danou stage. Postup přepnutí na jinou frontu bude následující:

1. Přijetí příkazu ke zpracování nové stage
2. Dokončení zpracování aktuální úlohy
3. Ukončení zpracovávání úloh z fronty pro aktuální stage
4. Stažení konfigurace pro zpracování nové stage
5. Zahájení zpracování úloh z fronty pro novou stage

## 4.5 Watchdog – Monitorování a automatická regulace

Systém pro automatickou regulaci zpracování slouží k přepínání workerů mezi jednotlivými stage. Přepínání probíhá na základě množství požadavků zpracování ve frontě pro danou stage, rychlosti zpracování požadavků a priorit nastavených administrátorem. Pro tuto komponentu byl zvolen název *Watchdog* (*Hlídací pes*), protože hlídá workery, aby pracovaly správně. Činnost této komponenty zahrnuje:

- *Monitorování zpracování* – zahrnuje monitorování vytížení front v message brokeru a získávání statistik o rychlosti zpracování různých stage.
- *Výpočet priorit* – na základě monitorovaných informací tento systém vypočítává priority zpracování každé stage.
- *Přepínání workerů* – na základě priorit zpracování zajistí, že workery zpracovávají stage s nejvyššími prioritami.

Watchdog neinteraguje přímo s workery v tom smyslu, že by se k nim připojoval a komunikoval s nimi. Všechna komunikace probíhá přes komponentu zajišťující koordinaci a správu nastavení. Worker reaguje na změnu svého nastavení stejně, jako by ji provedl administrátor.

Plánování probíhá na základě aktuálního stavu systému bez toho, aby si watchdog musel „pamatovat“ jakákoliv lokálně uložená data z předchozího plánování. Výsledkem plánování je přepnutí workerů na fronty těch stage, které právě potřebují obsluhu. Aktuální priority a jakékoliv další mezivýsledky se nikam neukládají, protože je lze kdykoliv vypočítat znovu. Interval mezi výpočty priorit bude stanoven na základě testů během vývoje.

Jediná synchronizace, kterou je třeba provádět, je výběr hlavního watchdogu, který bude vypočítávat priority. To je nutné hlavně kvůli prevenci přepisování konfigurací workerů rychle po sobě a také snížení zátěže na stahování monitorovacích informací. Více watchdogů je dobré jen pro případ, že by hlavní watchdog byl nedostupný, nebo selhal. V tom případě jeho funkci převezme jiný.

### Výpočet priorit front a plánování činnosti workerů

Úlohy jsou do front řazeny dle typu, ne dle priorit. Plánování probíhá nad těmito frontami, spíše než nad konkrétními úlohami. K naplánování zpracování fronty je třeba vzít v úvahu dobu zpracování úlohy v dané frontě, počet dostupných workerů a administrátorem definované priority, atd. Zároveň je třeba zajistit, že úlohy budou zpracovány a nebudou v systému front čekat nekonečně dlouho. To je zajištěno dynamickou úpravou plánování na základě statistik předchozího zpracování a aktuálně dostupných zdrojů (workerů). Podobný postup plánování lze najít například v distribuovaných systémech jako *SGE* (*Sun Grid Engine*) [12].

Navržený vzorec pro výpočet *parametrické priority fronty* (*parametric priority* nebo $PP_q$) je následující:

$$PP_q = \left(\frac{L_q \times AMT_q}{NW_q + 1} + WT_q\right) \times (AP_q + 1). \tag{4.1}$$

Parametrická priorita je, jak název napovídá, založena na parametrech fronty, které jsou popsány níže.

- $L_q$ – Délka fronty, neboli aktuální počet zpráv ve frontě (Length).
- $AMT_q$ – Průměrná doba zpracování zprávy z této fronty (Average Message Time).
- $NW_q$ – Počet workerů pracujících na frontě (Number of workers).
- $WT_q$ – Čas čekání nejstarší zprávy ve frontě (Waiting Time).
- $AP_q$ – Administrativní priorita fronty (Administrative Priority).

První část závorky, tedy $\frac{L_q * AMT_q}{NW_q}$ udává čas zpracování fronty daným počtem workerů. K počtu workerů je ve vzorci přičtena jednička, aby byl brán v úvahu stav, kdy na frontě žádný worker nepracuje. Frontám se tedy zvyšuje priorita na základě počtu zpráv a doby, nutné ke zpracování zprávy. Čím více workerů na frontě pracuje, tím nižší je priorita. To pomáhá zamezit přepínání všech workerů naráz na jedinou, nejvíce prioritizovanou frontu. Při přibližně stejném vytížení front budou workery pracovat na frontách více rovnoměrně. Další součástí vzorce je $WT_q$, čas čekání nejstarší zprávy na zpracování. Tento čas je počítán od chvíle, kdy je ve frontě alespoň jedna zpráva a zároveň na ní nepracuje žádný worker. Tím že plánování prioritizuje vytížené fronty, je třeba zamezit tomu, aby zpráva čekala v nevytížené frontě nekonečně dlouho. Čím více času bude zpráva čekat, tím vyšší bude priorita fronty, až nakonec bude i tato zpráva zpracována. Při přepnutí workeru na frontu je tento čas vynulován a zůstává nulový po celou dobu, kdy je fronta zpracovávána alespoň jedním workerem. Poslední část vzorce je $AP_q$ – administrativní priorita. Tato priorita umožňuje administrátorovi upravit výpočet a prioritizovat konkrétní frontu. K této prioritě je přičítána jednička, aby výpočet s výchozími parametry nevyšel nulový. Fronta může mít nulovou prioritu jen ve dvou případech. Když se administrativní priorita nastaví na hodnotu $-1$. Nebo když ve frontě nejsou žádné zprávy.

Parametrická priorita je číslo $PP_q \geq 0$, které je následně přepočítáno do rozsahu $< 0, 1 >$ na tzv. *relativní prioritu* (*relative priority*, $P_q$). Ta je vypočítána takto:

$$P_q = \left(\frac{PP_q}{\sum_{i=1}^{n} PP_i}\right), \tag{4.2}$$

kde $PP_q$ je parametrická priorita aktuální fronty a $\sum_{i=1}^{n} PP_i$ je součet parametrických priorit všech front tzn. *celková priorita* (*total priority*).

Důvod pro výběr právě těchto parametrů, použitých v parametrické prioritě fronty, je, kromě zajištění, že zprávy budou zpracovány, potřeba zajistit efektivitu při zpracování. Jinými slovy, je žádoucí, aby workery pracovaly na zpracování zpráv a ne na něčem jiném – v tomto případě na rekonfiguraci, která jim zabere čas. Prioritizací vytížených front se zajistí, že se workery budou méně přepínat mezi frontami, které mají málo zpráv, a tím se sníží čas, který stráví stahováním a aplikací nových konfigurací. Nevýhodou je, že pokud bude jedna fronta zahlcena, a budou do ní neustále přibývat nové zprávy, ostatní fronty budou mít čím dál menší prioritu. Budou muset déle čekat na zpracování, což vede ke zvýšení celkové latence zpráv, které systémem prochází.

Worker během zpracovávání fronty musí být schopen zpracovat minimální definované množství zpráv, než bude znovu přepnut na jinou frontu. K tomu slouží parametr *unlock time* (*čas uvolnění*), který říká, kdy může být worker přepnut na jinou frontu. Tento parametr nastavuje workerům watchdog ve chvíli, kdy jim změní konfiguraci, parametr je dostupný ze systému pro koordinaci a správu konfigurací. Díky tomuto parametru může dojít i k situaci, kdy se změní priority front, ale nebude dostupný worker, který by se mohl přepnout a začít frontu s nejvyšší prioritou zpracovávat. V tom případě dojde k přepnutí workeru až při příštím plánování. Pokud worker zpracovává frontu, ve které nejsou žádné zprávy – tzn. nic nedělá – pak je tento parametr ignorován a worker může být přenastaven i před vypršením tohoto intervalu. Při přepínání workerů je postup následující:

1. Jsou-li v systému workery, které nic nedělají, jsou postupně přepnuty na nejvíce prioritní fronty, přičemž priority jsou přepočítány při každém přepnutí workeru.

2. Pokud jsou všechny workery vytíženy, postupuje se od nejméně prioritní fronty k nejvíce prioritní a hledá se worker, který se může přepnout. Přepne se právě jeden worker.

3. Pokud není žádný worker volný, stav systému se nemění.

# Kapitola 5

# Implementace systému

V této kapitole je popis implementace systému a jeho komponent. Jsou zde popsány vybrané knihovny a aplikace, které jsou k implementaci použity a taktéž stručné zdůvodnění proč tomu tak je.

## 5.1 Definice požadavku zpracování

Zde je uvedeno, jak je realizováno zpracování pomocí ocr, popsané v kapitole 4. Pro každou stage je definována fronta v message brokeru (systému front) a konfigurace v systému pro správu konfigurací. Pipeline definuje uživatel. Ten při vytváření požadavku zpracování vybere, kterými stage požadavek projde. Jednotka zpracování je definována jako stránka dokumentu. Tato stránka je zapouzdřena ve zprávě předané message brokeru. Tato zpráva je označována jako *požadavek na zpracování* stránky (*processing request*), případně je označována obecně jako *zpráva* nebo *úloha*. Definice požadavku zpracování je následující:

1. *uuid* – id požadavku,
2. *page_uuid* – id stránky dokumentu,
3. *priority* – priorita požadavku,
4. *start_time* – čas začátku zpracování požadavku (jeho zápis do message brokeru),
5. *processing_stages* – seznam stage (etap), kterými má být stránka zpracována,
6. *results* – seznam výsledků zpracování,
7. *logs* – seznam logů; každý log odpovídá jedné stage, kterou požadavek prošel.

Pipeline je definována v *processing_stages*. Předání zprávy do další stage v pipeline zajišťuje worker, který zpracoval aktuální stage. Stránka dokumentu je ve zprávě mezi výsledky – obvykle první položka. Další výsledky mohou být xml soubor s přepisem dokumentu a metadata, která generuje OCR. Výsledky zpracování jsou zapsány ve zprávě v poli *results*. Mají formát, který se podobá souboru, aby je bylo možné jednoduše uložit. Formát obsahuje:

1. *name* – jméno souboru,
2. *content* – obsah souboru.

Poslední položkou jsou logy, kam worker zapisuje postup zpracování. Jsou zde zprávy z OCR knihovny a dalších knihoven a funkcí použitých při zpracování. Každý *stage log* (*log etapy zpracování*) obsahuje následující informace:

1. *host_id* – id workeru, který provedl zpracování dané stage,
2. *stage* – jméno stage,
3. *start* – čas začátku zpracování stage,
4. *end* – čas konce zpracování stage,
5. *status* – celkový výsledek zpracování, obsahuje řetězec *OK* nebo *Failed*,
6. *log* – zprávy zapsané do logu během zpracování.

Pro implementaci byl zvolen formát *Protobuf* (*Protocol Buffers*), který je detailněji popsaný v kapitole 3.4. Důvodem bylo jednoduché použití v jazyce python a dobrá dokumentace. Protobuf je také hlavní formát pro přenos dat používaný společností Google [8]. Díky tomu je zde předpoklad, že bude nadále rozvíjen a udržován. Formát *Flatbuffer*, zmíněný ve stejné kapitole, nemá plně implementované API pro jazyk python a tudíž nenabízí mnoho funkcí navíc. Proto byla tato volba učiněna spíše na základě preferencí a zmíněné parametry činí z formátu Protobuf dobrého kandidáta.

## 5.2 Systém front

Jak bylo popsáno v návrhu v kapitole 4.2, systém front je realizován pomocí message brokeru. Message brokery, které byly pro tento účel porovnávány, jsou popsané v kapitole 3.1.

K implementaci byl vybrán broker *RabbitMQ*. Broker splňuje požadavky, které jsou popsány v kapitole 4.2. Další důvody pro výběr právě tohoto brokeru jsou velmi dobrá dokumentace, která obsahuje detaily fungování, a tutoriál pro použití brokeru. Dostupný tutoriál zahrnuje i jazyk python, což bylo také plus.

RabbitMQ je použit v kombinaci s Dockerem, zmíněným v kapitole 3.3. Docker je použit k izolaci běhového prostředí a také k jednoduchému zprovoznění brokeru. Z kontejnerového registru Docker Hub, zmíněného v téže kapitole, je dostupný oficiální kontejner s nejnovější verzí RabbitMQ. Výhodou je, že broker není nutné instalovat, a kontejner přichází s již předpřipravenou základní konfigurací. To usnadňuje nasazení brokeru v produkci i testování během vývoje ostatních komponent.

K interakci s brokerem z jazyka python slouží knihovna *pika*, zmíněná v kapitole 3.1. Knihovna je implementována speciálně pro podporu RabbitMQ, ale umožňuje komunikaci i s jinými brokery. Komunikace probíhá pomocí protokolu *AMQP* (*Advanced Message Queuing Protocol*), ve verzi *0–9–1*. Protokol splňuje požadavky uvedené v návrhu. Z pohledu komunikace se jej týká zejména spolehlivé doručení a spolehlivé přijetí zprávy. Knihovna umožňuje i definování a mazání front a další práci s brokerem. Také je zde implementováno jednoduché odeslání tzv. *x-arguments*, volitelných argumentů fronty, které definují její chování a vlastnosti.

Atributy front pro předávání úloh v systému jsou následující:

- *x–max–priority* s hodnotou 1 zajišťuje, že zprávy mohou mít dvě priority, 0 nebo 1, takže lze některé zprávy prioritizovat před ostatními.
- *durable* s hodnotou *True*. Parametr definuje, že fronta bude v systému dostupná i po odpojení klientů a nebude automaticky smazána.

## 5.3 Systém pro koordinaci a správu konfigurací

Tato komponenta je navrhována v kapitole 4.3. Technologie, které byly zvažovány, jsou popsány v kapitole 3.2. Během vývoje bylo zjištěno, že tato komponenta bude muset být rozdělena na dvě části.

První z těchto částí je služba, zajišťující koordinaci distribuovaného systému. Tuto funkci zajišťuje *Apache Zookeeper*, který byl zvolen na základě informací v kapitole zmíněné v úvodu. Splňuje požadavky popsané při návrhu systému. Navíc v této kategorii software neexistuje mnoho dostupných alternativ, které by fungovaly podobným způsobem.

Během vývoje se objevil ten problém, že Zookeeper umožňuje uložení pouze malých souborů. Maximální velikost jsou *4kb*. Tuto hodnotu lze zvýšit v nastavení – tím se ale snižuje rychlost synchronizace a odezvy. Konfigurace OCR se skládá z textových konfiguračních souborů, ale i dodatečných binárních dat. Tyto binární soubory mohou dosahovat velikosti i několik set *MB*, takže se do uzlu v Zookeeperu nevejdou.

Pro řešení byl přidán *FTP* (*File Transfer Protocol*) server, který pomáhá s uložením těchto souborů. Řešení pomocí FTP bylo vybráno zejména kvůli rychlé implementaci a jednoduchému rozšíření systému během vývoje. FTP server samotný neumožňuje replikaci dat ani redundanci, nicméně data lze synchronizovat mezi více servery pomocí externí služby, která je k tomuto určena. Vzhledem k tomu, že bylo uvažováno o nahrazení jinou technologií, byl výběr služby pro synchronizaci souborů vynechán.

Stejně jako pro RabbitMQ, je i pro Zookeeper dostupný kontejner v registru Docker Hub. I zde je tedy použit Docker, aby nebylo nutné Zookeeper instalovat na testovací a případně i produkční servery a byla zjednodušena jeho správa a nastavení. Pro FTP server byl použit kontejner[1] uživatele *garethflowers*, používající *vsftpd* server. Kontejner byl zvolen kvůli jednoduchému použití a také z důvodů odstranění nutnosti instalace FTP na testovacím stroji.

## 5.4 Implementační detaily workeru

Worker byl vyvinut v rámci tohoto projektu a byl implementován v jazyce python3. Důvodem je použití OCR knihoven vyvinutých v rámci projektu PERO. Tyto knihovny jsou implementovány v pythonu a bylo by nepraktické použít jiný jazyk. Proto je worker implementován ve stejném jazyce. Detaily návrhu jsou popsány v kapitole 4.4.

Worker ke své činnosti potřebuje seznam Zookeeper serverů, které tvoří systém pro koordinaci a distribuci konfigurací. Seznam je mu předán přes parametr při startu. Adresy FTP serverů a RabbitMQ serverů si stáhne automaticky po připojení k Zookeeperu. Dále ke své činnosti potřebuje přístup k souborovému systému, aby si mohl dočasně ukládat soubory, potřebné ke konfiguraci OCR. Cesta se taktéž konfiguruje přes parametr, případně lze využít výchozí cestu `/tmp/pero-worker-id/`, kde id je identifikátor workeru. Identifikátor

[1] Použitý kontejner s FTP serverem: `https://hub.docker.com/r/garethflowers/ftp-server`.

je řetězec dodaný uživatelem, nebo vygenerované *uuid* (*Universal Unique Identifier*), kterým se worker registruje do systému pro koordinaci a správu konfigurací. Dále je zapotřebí zadat uživatelské jméno a heslo pro FTP server.

### Připojení workeru ke zbytku systému

Připojení k RabbitMQ zajišťuje knihovna pika. Pro práci se spojením bylo použito několik asynchronních funkcí, které jsou volány podle události, ke které dojde. Události, na které je reagováno jsou:

- Navázání spojení – vynuluje se čítač chyb při spojení.
- Chyba při navazování spojení – Navyšuje se čítač chyb a spojení se opakuje.
- Uzavření spojení – pokud je worker deaktivován, stav workeru se aktualizuje na vypnutý, jinak se worker snaží znovu připojit.

Spojení je multiplexováno pomocí tzv. *kanálu* (*channel*). Ten zajišťuje, že není třeba vytvářet více spojení a zabírat sockety v operačním systému, pokud aplikace navazuje více spojení s message brokerem, například kvůli zpracování více front zároveň. Namísto toho se vytvoří další kanál přes existující TCP/IP spojení. Kanál se taktéž vytváří pomocí asynchronních funkcí, ty se ale volají přímo po sobě a slouží k nastavení parametrů kanálu. Připojení neumožňuje automatické přepnutí na jiný server při výpadku. Proto je toto přepínání implementováno v rámci funkce pro navázání spojení. Při selhání připojení nebo při výpadku spojení se worker pokusí připojit na další message broker v seznamu.

Připojení k FTP probíhá během rekonfigurace workeru. Po stažení souborů pro aktuální stage se worker od FTP odpojí. Na rozdíl od spojení se Zookeeperem a RabbitMQ, toto spojení je aktivní jen po krátkou dobu, kdy je potřeba stáhnout soubory. Stejně jako u spojení s RabbitMQ, použitá knihovna, v tomto případě *ftplib*, neumožňuje automatické připojení k jinému serveru při výpadku. Proto je i zde implementována funkce pro připojení, která postupně vyzkouší všechny dostupné FTP servery.

Záznamy serverů pro message broker a FTP jsou typu *dictionary*. Tato struktura umožňuje rychlý výběr atributu pomocí klíče. Často se zapisuje jako JSON. Adresa a port serverů jsou při přijetí nových záznamů validovány a následně uloženy zvlášť kvůli jednoduchému použití. Příklad záznamu pro message broker:

`{"ip": 127.0.0.1, "port": 5672}`.

U Zookeeperu tento formát použit není a servery jsou uloženy v řetězci s formátem ve tvaru `adresa:port`, oddělené čárkami. Připojení k Zookeeperu je spravováno knihovnou *kazoo*, která automaticky přepíná mezi servery, pokud je některý z nich přetížený nebo nedostupný. U tohoto spojení tedy nejsou třeba pomocné funkce.

### Konfigurace a ovládání workeru

Worker je ovládán a konfigurován pomocí Zookeeperu, který mu předává novou konfiguraci pomocí volání tzv. *callback* funkcí. Callback je zpětné volání funkce, která aktualizuje lokální konfiguraci a tím ovlivňuje činnost workeru. Worker si po připojení k Zookeeperu vytvoří několik uzlů, které tvoří jeho konfiguraci, případně slouží k monitorování. Vytvoření této konfigurace je tu někdy označováno jako registrace workeru. Prefix cesty těchto polí je `/pero/worker/status/worker_id/`, kde `worker_id` je specifické pro daný worker. Pole konfigurace jsou:

- `status` – stav workeru (při registraci vždy `STARTING`),
- `queue` – fronta, kterou worker zpracovává, při registraci prázdné,
- `enabled` – povolení činnosti workeru, při nastavení na `false` se worker vypne,
- `unlock_time` – čas, kdy je možné worker přepnout na jinou frontu (worker toto pole vytvoří, ale nepoužívá, slouží k plánování viz. kapitola 4.5).

Worker aktualizuje pole `status`, které může nabývat hodnot

- `STARTING` – worker se spouští,
- `PROCESSING` – worker zpracovává zprávy z fronty,
- `RECONFIGURING` – worker mění konfiguraci (při přepnutí na novou frontu),
- `IDLE` – worker nic nedělá,
- `FAILED` – worker selhal,
- `DEAD` – worker neběží (byl vypnut).

Následující pole `queue` a `enabled` konfiguruje uživatel, aby upravil činnost workeru. Postup při změně fronty je popsán v kapitole 4.4. Tento postup je implementován v metodě `zk_callback_switch_queue`.

Další konfigurace mají prefix `/pero/worker/config/`. Tyto konfigurace jsou globální a používá je každý worker. Jsou zde pole:

- `mq_servers` – seznam RabbitMQ serverů,
- `ftp_servers` – seznam FTP serverů.

Dále worker reportuje statistiky a stahuje konfigurace pro specifickou frontu. K tomu používá pole s prefixem `/pero/queue/queue_name/`, kde `queue_name` je jméno zpracovávané fronty. Používaná pole jsou:

- `config` – textová konfigurace pro OCR,
- `config_path` – cesta k dodatečným souborům na FTP serveru,
- `avg_msg_time` – průměrný čas zpracování (je používán pro plánování, worker tento parametr aktualizuje),
- `avg_msg_time_lock` – zámek pro synchronizaci aktualizace průměrného času zpracování.

Zookeeper poskytuje atomické operace čtení i zápisu. Zde je zámek použit kvůli čtení a zápisu, které musí proběhnout po sobě. Worker při aktualizaci čte aktuální hodnotu a na jejím základě a na základě svych statistik počítá novou, kterou zapíše zpět. Zámek zamezuje *race condition* (*souběhu*) mezi čtením a zápisem, kdy by dva workery mohli aktuální hodnotu přečíst rychle po sobě a následně zaktualizovat bez zahrnutí výsledků toho druhého.

### Zpracování zpráv přijatých z RabbitMQ

Požadavky zpracování jsou, stejně jako změny konfigurace, přijímány pomocí callbacku. Tento callback reaguje na data poslaná z message brokeru. Zpráva s požadavkem je po přijetí parsována z Protobuf formátu. Data jsou následně předána ke zpracování, které probíhá podle konfigurace pro stage, ke které fronta patří. Součástí zpracování jsou obvykle soubor se zpracovávanou stránkou dokumentu, xml soubor s přepisem tohoto dokumentu a případně binární soubor s metadaty, který je používán OCR. Po ukončení zpracování jsou soubory ve zprávě aktualizovány, je přiložen log zpracování, a následně je zpráva odeslána ke zpracování další stage. Po odeslání úlohy worker čeká na přijetí potvrzení od message brokeru. Až potom sám odesílá potvrzení o zpracování zprávy, aby ji message broker odebral z jeho vstupní fronty. Pokud broker nepotvrdí přijetí, znamená to, že odeslání selhalo a worker se pokusí zprávu odeslat do výstupní fronty pipeline, což je poslední fronta v pipeline tohoto požadavku zpracování. Pokud se ani to nepodaří, worker odmítne danou zprávu, což způsobí, že ji dostane ke zpracování jiný worker. Tento postup slouží k odstranění problému s komunikací mezi workerem a message brokerem. Zde je ale nutný předpoklad, že pipeline daného požadavku zpracování je správně definovaná. To musí zajistit uživatel, či zadávající aplikace. Špatná definice pipeline způsobí zaseknutí zprávy s požadavkem ve frontě poslední správně definované stage, ze které ji nejde odeslat dál. Při selhání zpracování některé stage je zpráva taktéž poslána do výstupní fronty. Tím se přeskočí všechny následující stage, které nemá smysl zpracovávat, protože nejsou dostupná potřebná metadata v souborech zprávy.

Worker podporuje tzv. *dummy processing* či *dummy zpracování* (*simulované zpracování*). Stage může být definována jako dummy stage, což spočívá v tom, že namísto skutečného zpracování worker pouze čeká nějakou dobu, aby zpracování simuloval. Pro zapnutí dummy zpracování je třeba v konfiguraci OCR dané stage definovat pole `DUMMY = True`. Zde je vzorec pro výpočet doby zpracování (*processing time*): $PT = DS \times TS + TD$. $DS$ neboli *Data Size*, je délka dat (v bytech) prvního souboru v seznamu souborů zprávy. V konfiguraci je možné definovat hodnotu `TIME_SCALE`, zde $TS$, pro modifikaci tohoto času. Dále je zde hodnota $TD$, neboli *time delta* udávající odchylku. Tato hodnota se počítá jako náhodné číslo z intervalu s rovnoměrným rozložením pravděpodobnosti. V konfiguraci je interval definován jako `TIME_DIFF_MIN` a `TIME_DIFF_MAX`, nebo pomocí `TIME_DELTA`, kde je pak interval v rozmezí $< -TIME_DELTA, TIME_DELTA >$. Čas zpracování samozřejmě nemůže být záporný. Pokud tak vyjde, je nastaven na 0.

Pokud během zpracování dojde k přepnutí stage, kterou má worker zpracovávat, zpracování aktuální zprávy doběhne až do konce. Poté zůstane callback přijímající data zastavený na *zámku* (*lock* nebo *mutex*), který blokuje zpracování další zprávy. Po ukončení konzumace zpráv z fronty staré stage je zámek uvolněn a callback se dokončí. Další zprávy už nejsou přijímány, takže se callback aktivuje znovu až po změně konfigurace a zaregistrování workeru u fronty nové stage.

## 5.5 Implementační detaily watchdogu

Stejně jako worker, je watchdog vyvinut v rámci této práce. Je taktéž naprogramován v jazyce python3. Detaily návrhu a činnosti watchdogu jsou popsány v kapitole 4.5.

Informace o frontách se stahují ze systému pro správu konfigurací – tedy ze Zookeeperu, kam workery reportují statistiky o průměrné době zpracování. Také je zde zapsán čas, jak dlouho fronta čeká na zpracování, a její administrativní priorita. Následně jsou staženy informace o aktuálním počtu zpráv ve frontách. Tyto informace jsou stahovány z pluginu

pro správu RabbitMQ (*management plugin*), který umožňuje připojení pomocí HTTP. Ke stažení je použita knihovna *requests*, která poskytuje jednoduché API k vytváření HTTP dotazů. Data jsou obdržena ve formátu json. Je z nich použita pouze informace o počtu zpráv v každé frontě. Nakonec jsou ze Zookeeperu získány informace o stavu připojených workerů. Získané informace zahrnují stav workeru, frontu, kterou zpracovává a čas, kdy bude možné worker přepnout na novou frontu. Následně se vypočítají priority front podle vzorce v kapitole 4.5. Na základě této priority se upraví konfigurace workerů – worker z nejméně prioritní fronty se přepne na nejvíce prioritní. Poté se nastaví čas čekání na zpracování na aktuální čas pro všechny fronty, které obsahují zprávy, ale nezpracovává je žádný worker. Všechny tyto změny se provádějí v lokálních datech této instance watchdogu. Při provedení změny je v modifikované struktuře, která reprezentuje frontu nebo worker, nastaven parametr `modified` na `True`. Následně, po dokončení všech úprav, jsou změny naráz aplikovány v Zookeeperu, který zajišťuje distribuci nových konfigurací na jednotlivé workery.

Přepnutí workeru probíhá tak, že je v jeho konfiguraci upravena fronta, na které pracuje. K přepnutí může dojít, jen pokud worker pracuje na frontě, která je prázdná, nebo je jeho tzv. *unlock_time* (viz. konfigurace workeru 5.4) menší než aktuální čas. Čas, kdy je možné worker přepnout, je nastaven tak, aby worker stihl zpracovat minimální definovaný počet zpráv. Výpočet *unlock_time* workeru je následující: $UT_w = AMT_q \times MMN + MRT$, kde $AMT_q$ je *Average Message Time* – průměrný čas zpracování zprávy z dané fronty, uvedený v návrhu watchdogu v kapitole 4.5. $MMN$ je *Minimal Message Number* (*Minimální počet zpráv*, které musí worker zpracovat). Toto číslo bylo během experimentování nastaveno na 10. Poslední parametr $MRT$ označuje *Minimal Reconfiguration Time* (*Minimální čas nutný pro přenastavení*). Tento čas byl experimentálně nastaven na 10 a udává, jak dlouho workeru průměrně trvá změna konfigurace při přepnutí mezi forntami. Výsledkem tohoto vzorce je doba v sekundách, za jak dlouho je možné worker přepnout, aby mezitím stihl zpracovat alespoň deset zpráv. K výsledku je přičten aktuální čas a je přepočítán na pevně danou časovou značku ve formátu ISO-8601. Tato značka je následně zapsána do Zookeeperu.

Watchdog nebyl zcela dokončen. Chybí implementace jeho zaregistrování v Zookeeperu, aby bylo možné vybrat hlavní worker. Pokud bude běžet více watchdogů naráz, budou zbytečně zatěžovat systém redundantním stahováním statistik. Vzhledem k tomu, že workery mohou být přepnuty až po zpracování minimálního počtu zpráv, neměl by mít počet watchdogů zásadní vliv na dobu a počet přepnutí.

## 5.6 Ovládací skripty

Jak bylo uvedeno v kapitole 4.1, v rámci vývoje bylo vytvořeno i několik ovládacích skriptů. Jak bylo zmíněno v téže kapitole, předpokládá se, že vyvinutý systém bude integrován s demonstrační aplikací PERO. Tato aplikace poskytuje webové rozhraní pro jednoduché použití uživatelem. Vyvinuté skripty jsou určeny především pro administrátora systému a umožňují základní správu a monitorování systému z příkazového řádku. Druhá část skriptů jsou nástroje vyvinuté pro testování systému, které umožňují generování testovacích dat, měření statistik, apod.

První vyvinutý skript je *controller.py*, který slouží k ovládání workerů a získávání informací o stavu systému. Skript umožňuje vypsat seznam běžících workerů, jejich stav a frontu, na které pracují. Dále umožňuje uživateli přepínat workery mezi frontami, vypínat workery a odebírat ze systému vypnuté workery. Není zde dokončena implementace získání stavu front z RabbitMQ a spuštění uživatelem definovaného příkazu v Zookeeperu. Tyto

funkce nebyly implementovány, protože jde jen o zjednodušení přístupu k těmto informacím pro uživatele. Při vývoji byla dána přednost práci na workeru a již nezbyl čas na jejich dokončení. Statistiky front z RabbitMQ lze získat přihlášením do monitorovací konzole [24]. Příkazy v zookeeperu lze spouštět pomocí nástroje *zkCli.sh*, zmíněného v dokumentaci Zookeeperu [19].

Dále byl vyvinut nástroj *config_manager.py*. Tento skript slouží pro více obecné nastavení systému, které se netýká přímo jeho aktuálního stavu. Umožňuje definici nové stage, což spočívá ve vytvoření fronty v RabbitMQ a vytvoření konfiguračních polí v Zookeeperu. Dále umožňuje nahrát do Zookeeperu konfiguraci pro OCR, která definuje postup zpracování dokumentu v této stage. Také je zde možnost úpravy administrativní priority pro frontu dané stage a smazání stage. Poslední důležitou částí je možnost konfigurace serverů. Lze definovat message broker servery a FTP servery pro použití workerem. Také jsou zde message broker monitoring servery, které používá watchdog pro stahování informací o frontách.

Další vyvinutý nástroj je *publisher.py*, který, jak název napovídá, umožňuje publikovat zprávy (dokumenty) do message brokeru. Lze definovat seznam *images* (*obrázky*), kde každý „obrázek“ je dokument ke zpracování. Případně lze zadat cestu ke složce s dokumenty, které mají být nahrány do message brokeru. Také lze definovat pipeline, kterou budou dokumenty zpracovány, pomocí seznamu *stages*. Skript také umožňuje dokumenty stáhnout z výstupní fronty do lokální složky. Důležitou vlastností je možnost měření statistik a také uložení celých přijatých zpráv, aby z nich mohly být vytvořeny statistiky později, nebo mohly být použity k „debugování“ zpracování.

Další nástroj používaný i v rámci *publisher.py* je *stats.py*. Tento skript umožňuje měření statistik z přijatých zpráv. Statistiky mohou být měřeny přímo za běhu *publisher.py* v režimu stahování zpráv, nebo z již uložených zpráv. Nástroj umí v přijatých zprávách měřit dobu průchodu systémem a jednotlivými stage, zároveň umí detekovat pipeline a měřit průměrnou dobu zpracování v této pipeline.

Posledním důležitým nástrojem je *dummy_msg_generator.py*, který umožňuje generovat data pro dummy zpracování ve workeru, viz. 5.4. Nástroj vznikl k rychlému testování workeru, co se týče připojení k message brokeru a „debugování“ přepínání front. V kombinaci se *stats.py* je také velmi užitečný pro testování watchdogu, konkrétně chování funkce pro výpočet priorit front, protože umožňuje simulovat různé zatížení systému. K tomuto účelu lze definovat velikost zpráv, které určují dobu zpracování. Zároveň lze zadat interval odchylky od této doby, takže zprávy budou generovány z intervalu minimální a maximální velikosti. Také lze zadat počet opakování a periodu opakování zaslání zpráv.

Poslední dostupný nástroj je *aux_publisher.py*. Jde o první vyvinutý testovací nástroj pro worker. Vygeneruje jednu zprávu s dokumentem ke zpracování a čeká na její přijetí, poté ji uloží do výstupní složky a ukončí se.

# Kapitola 6

# Testování

Během vývoje bylo provedeno několik testů systému. Účelem testování bylo, kromě detekce chyb, i zhodnocení návrhu funkce pro výpočet priorit a vyzkoušení celkového chování systému. Test není proveden na základě skutečných dob, které jsou potřeba pro zpracování dokumentu pomocí OCR. Doby zpracování byly nastaveny tak, aby korelovaly s defaultními časovými limity pro různé parametry spojení, které bylo třeba nastavit a otestovat. Pro tento účel byl použit skript *dummy_msg_generator.py*, který slouží pro generování testovacích zpráv a je popsaný v kapitole 5.6. Statistiky byly vygenerovány pomocí skriptu *stats.py* po stažení zpráv ze systému. Veškeré měření probíhalo lokálně na jednom počítači, latence přenosu po síti tedy byly nulové. Měření se soustředilo především na měření latence zprávy při průchodu systémem, tedy na celkovou dobu, kterou zpráva v systému stráví, než je předána s výsledkem zpět uživateli. Hlavní myšlenkou je zde zjištění závislosti doby čekání ve frontě na celkovém zatížení systému. Je zřejmé, že pokud bude ve frontě větší množství zpráv, bude doba průchodu zprávy systémem vyšší, než pokud přijde hned na řadu, protože je fronta prázdná. Během testu byly v systému nadefinovány tři stage, které tvořily jedinou pipeline. Konfiguraci popisuje tabulka 6.1.

Průběh testu byl takový, že po nastavení systému byly přidány 2 workery a watchdog. Následně byly generovány testovací zprávy, vždy v sérii po pěti. Další série zpráv byla vygenerována po 220 sekundách, což je přibližně dvojnásobek doby, nutné pro zpracování jedné zprávy. Díky tomu se v systému postupně hromadily zprávy, čímž byla negativně ovlivněna doba jejich zpracování.

Ze statistiky pro test 1 v tabulce 6.2 je vidět, že se zprávy hromadily ve frontě druhé stage. Díky tomu byl průměrný čas čekání ve frontě této stage vyšší. To podporuje tvrzení, že funkce pro výpočet priorit preferuje více vytížené fronty, viz. návrh funkce pro výpočet priorit, kapitola 4.5. Jak je vidět z tabulky, první a třetí stage mají zároveň vyšší dobu zpracování, což taktéž vede k vyšší prioritě během zpracování. Vysoká doba čekání u první stage

| stage | max. doba zpracování | min. doba zpracování | průměrná doba zpracování |
|---|---|---|---|
| stage1 | 56 | 36 | 46 |
| stage2 | 48 | 29 | 38,5 |
| stage3 | 38 | 8 | 23 |

Tabulka 6.1: Nastavení rychlosti zpracování pro jednotlivé stage, použité při testování. Uvedené doby jsou v sekundách.

| Číslo testu | stage1 | | stage2 | | stage3 | | Celkem |
|---|---|---|---|---|---|---|---|
| | Zpracování | Čekání | Zpracování | Čekání | Zpracování | Čekání | |
| 1 | 45,24 | 524,76 | 22,62 | 1366,73 | 30,54 | 238,78 | 2228,70 |
| 2 | 45,02 | 722,06 | 22,51 | 1023,62 | 29,03 | 144,82 | 1987,07 |
| 3 | 45,31 | 947,21 | 22,65 | 1130,28 | 31,26 | 143,18 | 2319,90 |
| 4 | 44,90 | 413,34 | 22,45 | 469,65 | 22,57 | 566,64 | 1539,55 |
| 5 | 45,32 | 526,33 | 22,66 | 674,72 | 23,21 | 558,71 | 1850.94 |
| 6 | 44,91 | 375,58 | 22,45 | 769,67 | 22,35 | 717,70 | 1952,68 |
| 7 | 45,00 | 376,68 | 22,50 | 539,08 | 22,59 | 450,18 | 1456.03 |

Tabulka 6.2: Tabulka všech výsledků testů. Jsou zde doby čekání a zpracování všech stage. Doby jsou uvedené v sekundách. Je zde vidět, že celková rychlost průchodu zpráv systémem je nejvíce ovlivněna nerovnoměrnou dobou čekání na zpracování v různých stage. Při přibližně stejné době čekání na zpracování všech stage, je průchod zprávy systémem rychlejší a rovnoměrnější.

byla nejspíše způsobena novými zprávami, přicházejícími do systému, které byly nahrávány do této fronty.

Další test cílil na zlepšení rychlosti průchodu systémem. K tomuto účelu byly použity administrativní priority front, které byly nastaveny na pozdější fáze zpracování. Priorita první stage zůstala nulová, pro stage2 a stage3 byla priorita zvýšena o jedna. Výsledky popisuje řádek 2 v tabulce 6.2. Test 3 proběhl podobně, priorita stage3 byla zvýšena o jedna. Cílem bylo zajistit vyšší prioritu zpracování pro úlohy, které jsou již rozpracované. Stage, které byly dále ke konci pipeline, dostaly větší prioritu, aby bylo preferováno dokončování rozpracovaných úloh v pipeline. Nicméně statistiky pro tento test vyšly hůře než pro předchozí test.

Tabulka 6.2 zobrazuje statistiky pro všechny provedené testy. Je zde vidět vliv poslední stage na celkovou dobu zpracování. Zatímco u ostatních stage se tato doba příliš nemění, u této kolísá zásadním způsobem. V testu číslo 3, kde je třetí stage průměrně nejpomalejší ze všech provedených testů, lze zároveň pozorovat dlouhou celkovou dobu zpracování, i dlouhou dobu čekání ve frontě předchozích stage. Naopak v testu číslo 7, kde je doba zpracování ve stage3 téměř nejnižší, lze pozorovat přibližně stejné doby čekání ve frontách všech stage. Zároveň je celková rychlost zpracování mnohem vyšší, pravděpodobně kvůli rovnoměrnější práci workerů na všech stage.

Zajímavý je taktéž fakt, že změna administrativních priorit neměla příliš velký vliv na rychlost zpracování. Konfigurace administrativních priorit byly v testech 1, 6, 7 nastaveny na nule, v testech 2 a 5 byly zvýšeny o jedna pro fronty stage2 a stage3 a ve zbývajících případech, v testech 3 a 4 se postupně zvyšovaly. Tedy stage1 měla prioritu 0, stage2 dostala prioritu 1 a stage3 prioritu 2. Jak je vidět, nejrychlejší byl test 7, kde priority na frontách nastaveny nebyly, ale hned druhý nejrychlejší je 4, kde se priority postupně zvyšovaly. Důvodem byla nejspíše příliš nízká nastavená priorita, vzhledem k počtu zpráv a rychlosti zpracování. Pokud by priority byly navýšeny o desítky bodů, namísto jednotek, byl by jejich vliv větší.

Z naměřených statistik lze pozorovat, že plynulost zpracování má zásadní vliv na rychlost průchodu zprávy systémem. Toto lze vidět v testu 7, kde zprávy čekají ve všech frontách přibližně stejně dlouho. Zároveň má velký vliv doba zpracování zprávy v dané stage, což je dáno důrazem na tento parametr ve funkci pro výpočet priorit. Díky přibližně stejně dlouhé době zpracování mezi stage2 a stage3 v testu 7, měly pravděpodobně tyto fronty přibližně

stejnou prioritu. To by podporovalo teorii, že workery zpracovávaly tyto stage rovnoměrně. Naopak v ostatních testech, zejména pak prvních třech, byly přepínány oba workery naráz na jednu stage a zpracování díky tomu probíhalo více nárazově.

Pro potvrzení této teorie by v budoucích testech bylo třeba zahrnout časovou stopu workerů, kde bude vidět, ve který čas pracovaly na které stage. Zároveň by bylo vhodné přidat konkrétní informace o vytížení front v čase. Tyto informace by mohl reportovat například watchdog, který zajišťuje přepínání workerů. Výsledkem by mohla být data potřebná k návrhu lepší plánovací funkce, která zajistí rovnoměrnější zpracování a větší propustnost systému. Úprava by mohla zahrnovat větší důraz na množství zpráv ve frontě a nižší důraz na dobu zpracování dané stage. Zároveň je ale třeba vzít v úvahu, že jde jen o testovací data, nejde o skutečné rychlosti zpracování, které by byly naměřeny za provozu systému. Tam bude záležet i na rychlosti OCR a její závislosti na velikosti dokumentu, době načtení konfigurací, která je zde nastavena experimentálně na 10 sekund, a v neposlední řadě na latenci síťového provozu. V každém případě lze tyto informace použít k lepšímu nastavení administrativních priorit, které mohou pomoci vyvážit rozdíly v prioritách front i bez změny plánovací funkce.

## 6.1 Testování přepisu dokumentu

Po dokončení workeru byly provedeny testy na skutečných datech. Vývoj ani konfigurace OCR nejsou součástí této práce, testy byly provedeny na konfiguracích dodaných členy projektu PERO. Cílem bylo zjistit, zda zpracování a rekonfigurace workeru při přepnutí fronty probíhá správně. Výsledkem testu byl úspěšný přepis dokumentů do XML. Dokumenty byly v tomto případě PNG obrázky[1], s naskenovaným textem ve stylu psacího stroje. Pro zajímavost – doby zpracování použitých dokumentů byly někde mezi 1 a 4 sekundami pro všechny stage. Nicméně tato doba závisí na velikosti obrázku a náročnosti detekce a extrakce textu. K naměření relevantních statistik bude v praxi potřeba blíže spolupracovat s týmem projektu PERO kvůli výběru relevantních dat pro tento test. Účelem tohoto testu bylo především ověření funkčnosti systému, ne hodnocení jeho výkonu. Vzhledem k tomu, že testy probíhaly na jednom testovacím stroji, výsledky měření by byly ovlivněny spíše zatížením tohoto stroje a nereflektovaly by realitu. Výkonnostní testy jsou plánovány na dobu po zprovoznění systému na výpočetních serverech projektu PERO.

[1] Příklad obrázku použitého při testech přepisu dokumentu do XML: https://commons.wikimedia.org/wiki/File:Text_entropy.png.

# Kapitola 7

# Závěr

Cílem této práce byl návrh a implementace robustního a distribuovaného systému pro přepis dokumentů. Tento cíl byl splněn a výsledný systém je funkční a schopný zpracování dat.

Během návrhu systému byla prostudována existující řešení podobných problémů. Kapitola 3 obsahuje výtah nalezených technologií, které tyto problémy řeší. Z těchto technologií byl vybrán Apache Zookeeper pro koordinaci činnosti systému a RabbitMQ pro distribuci úloh. Detaily komponent realizovaných vybranými programy jsou uvedeny v návrhu systému v kapitole 4. Postup výběru je uveden v kapitole implementace 5. Komponenty realizující zpracování dokumentů pomocí OCR a plánování zpracování byly implementovány v rámci této práce.

Jsou k dispozici výsledky testování implementovaného systému, které si lze prohlédnout v kapitole 6. Též je zde uvedeno několik návrhů na zlepšení budoucích testů i výkonu systému.

Systém splňuje svůj hlavní účel, pro který byl připravován. V dalším vývoji je třeba provést doplnění chybějících funkcí, uvedených u popisu implementací jednotlivých komponent v kapitole 5. Doplněny budou části potřebné pro „ostrý“ provoz, mezi kterými lze jmenovat nedokončenou synchronizaci komponenty pro plánování zpracování a nedokončené administrátorské nástroje.

Další fází vývoje by mělo být zprovoznění systému na výpočetních serverech projektu PERO. Zde mohou být provedeny výkonnostní testy se skutečnými daty. Na základě výsledků lze případně naplánovat další úpravy systému. Následovat bude integrace do demonstrační aplikace projektu PERO.

Výsledný software je dostupný volně ke stažení ze stránek GitHubu[1].

[1] URL pro stažení výsledné aplikace: `https://github.com/DCGM/pero-worker`

# Literatura


[1] Apache ActiveMQ development team. *ActiveMQ 5 Documentation* [online]. The Apache Software Foundation, 2021 [cit. 2021-10-24]. Dostupné z: https://activemq.apache.org/components/classic/documentation.

[2] Apache ActiveMQ development team. *Apache ActiveMQ Artemis Documentation* [online]. The Apache Software Foundation, 2021 [cit. 2021-10-24]. Dostupné z: https://activemq.apache.org/components/artemis/documentation/.

[3] Bohmann, D. *Srovnání implementací message brokerů* [online]. Plzeň, 2020. [cit. 2021-10-22]. Disertační práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky. Dostupné z: https://dspace5.zcu.cz/bitstream/11025/41755/1/Bohmann_David_2020_DP.pdf.

[4] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* [online]. USA: Google Inc., 2006 [cit. 2022-04-12]. Dostupné z: https://research.google/pubs/pub27897/.

[5] Confluent Inc.. *Confluent kafka python* [online]. Confluent Inc., 2016 [cit. 2022-04-09]. Dostupné z: https://docs.confluent.io/platform/current/clients/confluent-kafka-python/html/index.html.

[6] FIT VUT v Brně. *PERO - Pokročilá extrakce a rozpoznávání obsahu tištěných a rukou psaných digitalizátů pro zvýšení jejich přístupnosti a využitelnosti* [online]. FIT VUT v Brně, březen 2018. 2021 [cit. 2022-04-04]. Dostupné z: https://www.fit.vut.cz/research/project/1165/.cs.

[7] Garnock Jones, T., Roy, G. M., Pivotal Software Inc. et al. *Introduction to Pika* [online]. ReadTheDocs.io, březen 2011 [cit. 2021-10-23]. Dostupné z: https://pika.readthedocs.io/en/stable/.

[8] Google Inc.. *Protocol Buffers*. Google Inc., 2008 [cit. 2021-12-03]. Dostupné z: https://developers.google.com/protocol-buffers.

[9] Google Inc.. *Flatbuffers*. GitHub, 2014 [cit. 2021-12-03]. Dostupné z: https://google.github.io/flatbuffers/.

[10] Heidi, E. *What is High Availability?* [online]. DigitalOcean, únor 2016. 2021 [cit. 2022-04-07]. Dostupné z: https://www.digitalocean.com/community/tutorials/what-is-high-availability.

[11] Jangla, K. *Accelerating Development Velocity Using Docker: Docker Across Microservices*. 1. vyd. USA: apress, 2018. ISBN 978-1-4842-3935-3.

[12] Oracle Corporation. *Scheduling Strategies*. Oracle Corporation, 2010 [cit. 2022-03-17]. Dostupné z: https://docs.oracle.com/cd/E19957-01/820-0698/chp9-22/index.html.

[13] Powers, D., Arthur, D. et al. *Kafka-python* [online]. ReadTheDocs.io, 2016 [cit. 2022-04-09]. Dostupné z: https://kafka-python.readthedocs.io/en/master/.

[14] RedHat, Inc.. *What is orchestration?* RedHat, Inc., říjen 2019 [cit. 2021-11-25]. Dostupné z: https://www.redhat.com/en/topics/automation/what-is-orchestration.

[15] Richardson, C. *Microservice Architecture*. Chris Richardson, březen 2014 [cit. 2021-12-21]. Dostupné z: https://microservices.io/.

[16] Solem, A. et al. *Celery – Distributed Task Queue* [online]. Ask Solem, 2009 [cit. 2021-10-23]. Dostupné z: https://docs.celeryq.dev/en/stable/index.html.

[17] TechTerms.com. *OCR* [online]. TechTerms.com, duben 2016 [cit. 2022-04-07]. Dostupné z: https://techterms.com/definition/ocr.

[18] Teng, Q., Bannister, T. et al. *Kubernetes*. The Linux Foundation, březen 2016 [cit. 2021-11-25]. Dostupné z: https://kubernetes.io/.

[19] The Apache Software Foundation. *ZooKeeper Documentation* [online]. The Apache Software Foundation, 2009 [cit. 2022-04-12]. Dostupné z: https://zookeeper.apache.org/doc/r3.7.0/index.html.

[20] The Apache Software Foundation. *Apache Zookeeper* [online]. The Apache Software Foundation, 2010 [cit. 2022-04-12]. Dostupné z: https://zookeeper.apache.org/.

[21] The Apache Software Foundation. *Apache Qpid Documentation* [online]. The Apache Software Foundation, 2015 [cit. 2021-10-27]. Dostupné z: https://qpid.apache.org/documentation.html.

[22] The Apache Software Foundation. *Kafka – Documentation* [online]. The Apache Software Foundation, 2017 [cit. 2022-04-09]. Dostupné z: https://kafka.apache.org/documentation/#gettingStarted.

[23] The ZeroMQ authors. *ZeroMQ – An open-source universal messaging library* [online]. The ZeroMQ authors, 2021 [cit. 2021-10-23]. Dostupné z: https://zeromq.org/.

[24] VMWare Inc.. *RabbitMQ – the most widely deployed open source message broker* [online]. VMWare Inc., 2007 [cit. 2021-10-22]. Dostupné z: https://www.rabbitmq.com/.