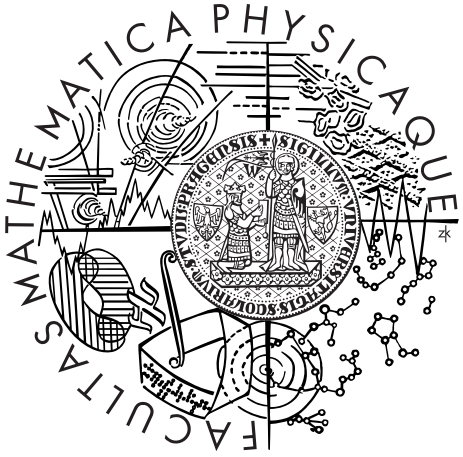
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Bc. Miroslav Tamáš

Webový vyhledávací systém

Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: RNDr. Leo Galamboš, Ph.D.
Studijní program: Informatika
Studijní obor: Softwarové systémy

Praha 2013

Na tomto mieste by som rád poďakoval predovšetkým svojmu vedúcemu diplomovej práce RNDr. Leovi Galambošovi, Ph.D. za odborné rady a smerovanie, ktoré mi pomohli pri návrhu a implementácii. Taktiež by som chcel poďakovať svojím rodičom a priateľke, ktorí mi boli vždy oporou.

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne a výhradne s použitím citovaných prameňov, literatúry a ďalších odborných zdrojov.

Beriem na vedomie, že se na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona v platnom znení, hlavne skutočnosť, že Univerzita Karlova v Prahe má právo na uzavretie licenčnej zmluvy o užití tejto práce ako školného diela podľa §60 odst. 1 autorského zákona.

V ..... dňa ..... Podpis autora

Názov práce: Webový vyhledávací systém

Autor: Bc. Miroslav Tamáš

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedúci diplomovej práce: RNDr. Leo Galamboš, Ph.D., Ústav bezpečnostních technologií a inženýrství, České vysoké učení technické v Praze

Abstrakt: Akademický fulltextový vyhľadávač Egothor sa v posledných rokoch stal základom viacerých prác z oblasti vyhľadávania. Doposiaľ však neexistovalo riešenie, ktoré by poskytlo kompletnú sadu nástrojov pre spracovanie webového obsahu vo väčšom merítku. Táto práca sa zaoberá návrhom a implementáciou distribuovaného vyhľadávacieho systému zameraného predovšetkým na internetové zdroje. Analyzuje komponenty prvej generácie systému pre spracovanie webového obsahu a predstavuje ich primárne funkcie. Následne popisuje ich využitie pri návrhu architektúry distribuovanej varianty webového vyhľadávača. Návrh sa zameriava predovšetkým na fázy získavania, spracovania a indexácie dát. Následne popisuje spôsob implementácie uvedeného riešenia. V závere potom predstavuje niekoľko návrhov ako na dosiahnuté výsledky nadviazať.

Kľúčové slová: galaxy, vyhľadávač, index, wayback, distribuované spracovanie, dataset, worker, procesor, j5m, crawler, egothor, konektor, webové služby

Title: Web Search System

Author: Bc. Miroslav Tamáš

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Leo Galamboš, Ph.D., Department of Security Technologies and Engineering, Czech Technical University in Prague

Abstract: Academic fulltext search engine Egothor has recently became starting point of several thesis aimed on searching. Until now, there was no solution available to provide robust set of web content processing tools. This master thesis is aiming on design and implementation of distributed search system working primary with internet sources. We analyze first generation components for processing of web content and summarize their primary features. We use those features to propose architecture of distributed web search engine. We aim mainly to phases of data fetching, processing and indexing. We also describe final implementation of such system and propose few ideas for future extensions.

Keywords: galaxy, search engine, index, wayback, distributed processing, dataset, worker, procesor, j5m, crawler, egothor, connector, web services

Obsah

Kapitola 1

Úvod

Objem informácií dostupných na Internete rastie každým dňom. Fulltextové vyhľadávače tak musia poskytovať čo najrelevantnejšie odpovede na užívateľské dotazy, a preto potrebujú toto obrovské množstvo vstupných informácií v prvom rade čo najrýchlejšie spracovať. Rýchlosť spracovania sa netýka výhradne oblasti veľkých internetových vyhľadávačov ako napríklad Google.com [1], Yahoo.com [2] či Bing.com [3], ktoré pokrývajú miliardy webových stránok a ich index narastá každým dňom tak, ako nepretržite narastá množstvo nových webových stránok. Prvý index spoločnosti Google obsahoval v roku 1998 podľa [4] zhruba 26 miliónov internetových stránok, pričom ich počet za ďalšie dva roky prekonal neuveriteľnú hranicu jednej miliardy. Vyhľadávanie sa v posledných rokoch nezameriava len na oblasť internetu. Stáva sa dôležitým faktorom aj vo firemnej či korporátnej sfére. Spoločnosti a organizácie na celom svete získavajú a spravujú obrovské množstvá informácii vo forme interných či verejných dokumentov, databázových záznamov a multimédií pochádzajúcich zo širokého spektra zdrojov. Problém vyhľadávania v takýchto informáciách sa preto stáva čoraz viac diskutovanejším. V súčasnosti existuje na trhu viacero rôznych softwareových riešení, ktorých primárnym cieľom je pokrytie požiadaviek nielen v oblasti Enterprise Search [5], ale celkovo v oblasti Open Source Intelligence [6], ďalej len OSINT. Medzi najznámejšie proprietárne riešenia patria napríklad Intelligent Data Operating Layer [7], v skratke IDOL, od spoločnosti Autonomy, ktorú zhruba pred rokom kúpila firma Hewlett Packard. V obasti Enterprise Search ale nezaostávajú ani české spoločnosti, medzi najznámejšie patrí firma Tovek s produktom Tovek Server [8]. K dispozícii sú taktiež open-source riešenia, medzi ktorým vedie predovšetkým platforma Apache Solr [9] postavená na vyhľadávacom stroji Lucene [10]. Všetky uvedené produkty predstavujú softvérové balíčky, ktorých primárnym cieľom je zber informácií z požadovaných dátových zdrojov, spracovanie, uloženie a indexácia do interných dátových štruktúr, za účelom poskytnutia možností dotazovania sa nad takto pripravenými dátami.

Táto diplomová práca sa zameriava predovšetkým na návrh a implementáciu prototypu softvérového systému určeného k spracovaniu dát z interných a externých zdrojov, tak aby užívatelia systému mali možnosť v takto sledovaných zdrojoch vyhľadávať požadované dáta. Výsledné riešenie bude navrhnuté a implementované nad komponentami prvej generácie systému pre spracovanie webového obsahu, ktoré vznikli výhradne na akademickej pôde.

1.1 Predstavenie problému

Obecný webový vyhľadávací systém je určený predovšetkým na vyhľadávanie informácii, a to nie len na internete, ale aj v iných zdrojoch, ako napríklad firemných intranetoch, či lokálnych dátových úložiskách. Samotnému procesu fulltextového vyhľadávania predchádza niekoľko krokov, bez ktorých by vyhľadávanie ako také nebolo možné. Skôr, ako môže užívateľ začať vyhľadávať v požadovanej kolekcii dát, musí systém v prvom rade tieto dáta získať a spracovať tak, aby užívateľovi poskytoval odpovede na dotazy v reálnom čase. Konkrétne, systém typu *Enterprise Search*, umožňujúci vyhľadávanie v dátových zdrojoch musí byť schopný vykonávať nasledujúce operácie podľa možností čo najefektívnejšie:

1. Vyťažovanie dát z požadovaného dátového zdroja
2. Spracovanie surových dát a indexácia
3. Vyhľadávanie

Akvizičná časť systému vyťažuje dátové zdroje. Dátovým zdrojom v kontexte tejto práce rozumieme akýkoľvek typ zdroja obsahujúci dokumenty, súbory či iné záznamy, v ktorých chce cieľový užívateľ využívať funkcie fulltextového vyhľadávania. Príkladom takýchto zdrojov sú webové stránky, emailové účty, diskusné fóra či transkripcia zvukových záznamov, prípadne akékoľvek iné zdroje obsahujúce záujmové informácie. Získané dáta systém následne uloží na lokálne úložisko, alebo odošle priamo k ďalšiemu spracovaniu, ktoré vo finále končí indexáciou.

Informácie, v ktorých užívateľ vyhľadáva môžu byť uložené v širokej škále rôznych dátových formátov. Proces spracovania surových dát prebieha predovšetkým z dôvodu potreby extrakcie textového obsahu z týchto formátov. Textová informácia tvorí základný vstup invertizačného procesu. Ďalšia extrakcia dodatočných informácií či ďalších meta-dát navyše umožní poskytnúť užívateľovi doplňujúce funkcie pre spresnenie požadovaného dotazu, napríklad filtrácia podľa jazyka, formátu, typu zdroja či iného. Posledným krokom spracovania je tak invertizácia a indexácia takto extrahovaných dát či ďalších dodatočných polí, respektíve tokenov.

Predchádzajúca úspešná indexácia získaných dát vo finále umožní užívateľovi zadávať nad zvolenou množinou dokumentov vlastné dotazy a rýchlo vyhľadať požadované dokumenty. Extrahované meta dáta je možné použiť k upresneniu požiadaviek na hľadaný dokument. Výsledková listina by mala obsahovať nielen štandardnú sadu informácií, akými sú názov dokumentu, referencia a krátky útržok textu, ale taktiež informácie o čase indexácie či poradovom čísle revízie daného dokumentu. Bohužiaľ, dáta dostupné v čase indexácie už nemusia nutne v dobe vyhľadávania existovať v pôvodnej podobe. Prípadne už nemusia existovať vôbec. Vyhľadávací systém preto musí umožniť užívateľovi náhľad na snapshot, respektíve kešovanú verziu pôvodných dát tak, ako boli k dispozícii v čase indexácie.

1.2 Požiadavky

Cieľom práce je preštudovať doposiaľ existujúce komponenty prvej generácie systému pre spracovanie webového obsahu, navrhnúť a implementovať distribuované riešenie plne funkčného webového vyhľadávacieho systému. Vstupom sú nasledujúce komponenty:

- *J5M* [11] - Java 5 Middleware, framework umožňujúci rýchly vývoj distribuovaných aplikácií
- *Egothor2* [12] - Fulltextový vyhľadávací stroj
- *WSE* [13] - Web Search Engine, standalone webový vyhľadávací systém prvej generácie.
- *BOBO* [14] - Distribuovaný robot
- *SUHA-COMMONS* [15] - API rozhranie k funkciám webového robota BOBO

Uvedené komponenty budú tvoriť základný kameň výsledného riešenia, ktoré by malo napĺňať nasledujúce požiadavky:

1. Management crawlovacieho procesu, teda vytváranie, ovládanie a správu vyťažovacích úloh pre robota BOBO [14]. Jedná sa predovšetkým o zadávanie úloh, vytváranie webových archívov, kde sa získané dáta ukladajú a vo finále indexácia takto preddefinovaných archívov.
2. Management filtrovacieho procesu, predovšetkým z pohľadu spracovania získaných dát a ich obohatenia o dodatočné meta dáta.
3. Management väčšieho počtu fulltextových indexov a predovšetkým možnosť vytvárania vlastných dátových kolekcií určených k vyhľadávaniu.
4. Správu užívateľských skupín s možnosťou definovania rôznych stupňov oprávnení pre prístup užívateľov k jednotlivým funkciám výsledného systému.
5. Priechod históriou získaných dát, tzv. wayback machine, kedy sa u všetkých dokumentov udržujú historické verzie dostupných dát.
6. Sledovanie spätných odkazov, inými slovami možnosť prekladu referencie na identifikátor dokumentu dostupného vrámci systému.
7. Vývoj vlastných aplikácií nad dostupným rozhraním webových služieb. Konkrétne v rámci návrhu vyhľadávacieho systému sa požaduje vytvorenie rozhrania webových služieb, ktoré by publikovalo všetky dostupné funkcie výsledného jadra systému tak, aby bolo v prípade potreby možné tieto funkcie začleniť do väčších informačných celkov. Navrhnuté rozhranie by malo taktiež podporovať možnosť napojenia výsledného riešenia do väčších informačných celkov s použitím BPEL technológií [17].
8. Podpora offline dotazovania. Konkrétne návrh aparátu, ktorý umožní implementovať komplexnejšie dotazovanie využívajúce predovšetkým informačnej vzdialenosti medzi zvolenými entitami.

1.3 Štruktúra práce

Formálne členenie textu diplomovej práce kopíruje spôsob, akým bola práca postupne riešená. V nasledujúcej kapitole 2 stručne popíšeme jednotlivé komponenty tvoriace základné stavebné kamene výslednej implementácie. Priblížime predovšetkým ich prvotný koncept a kľúčové vlastnosti využité pri riešení zadania. Kapitolu 3 venujeme popisu základných konceptov výsledného riešenia. Identifikujeme nielen hrubý koncept a jeho silné stránky ale taktiež ich dopad na budúci rozvoj celého systému. V kapitole 4 následne priblížime a popíšeme technickú realizáciu výsledného návrhu. Predposlednú kapitolu využijeme na priblíženie vízie ďalšieho rozvoja výslednej implementácie. V závere potom zhrnieme dosiahnuté výsledky s ohľadom na prvotné zadanie.

Kapitola 2

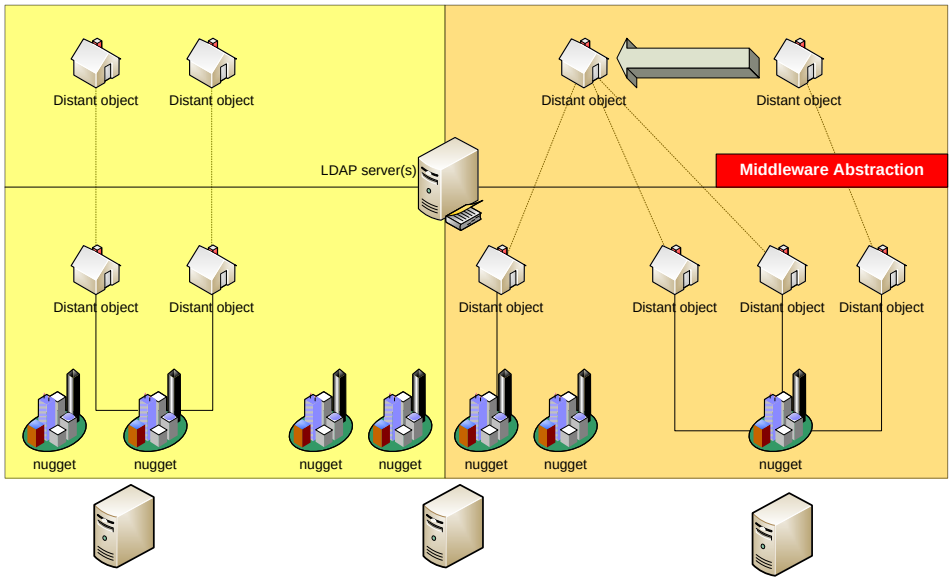
Prvá generácia komponent pre spracovanie webového obsahu

Kapitola predstaví komponenty prvej generácie pre spracovanie webového obsahu. Primárnym cieľom kapitoly je predovšetkým predstavenie riešení, ktoré tvoria základ pre implementáciu navrhovaného distribuovaného a škálovateľného vyhľadávacieho systému. Konkrétne, základom distribuovaného systému je J5M middleware [11], ktorý umožňuje nasadenie systému v rámci výpočtového klastru a poskytuje sadu funkcií tvoriacich abstrakciu nad RMI [18, 19]. Uvedený rámec sa stal základom, nad ktorým bol vytvorený distribuovaný robot BOBO [14]. Primárnym cieľom robota je zber dát z prostredia internetu, predovšetkým z webových stránok. Architektúra robota bola navrhnutá tak, aby bolo možné implementovať a nasadzovať rôznorodé techniky vyťažovania. Súčasťou robota je taktiež projekt SUHA-COMMONS [15], ktorý publikuje API rozhrania k jednotlivým subsystémom a dátovej vrstve robota. Získané, respektíve vyťažené dáta sú následne uložené do takzvaného fulltextového indexu, ktorý tvorí základnú časť vyhľadávacieho systému. Jedná sa o dátovú štruktúru, nad ktorou je možné dotazovanie s pomocou dotazovacieho jazyka. V rámci tejto práce, bola ako fulltextový index vyžadovaná knižnica Egothor2 [26], ktorá predstavuje multiplatformový vyhľadávací stroj pochádzajúci z akademického prostredia. Poslednou predstavovanou komponentou, projekt Web Search Engine [13], je prvá generácia integrovaného užívateľského rozhrania pre správu vyťažovacieho a indexačného procesu. Komponenta zároveň slúži ako vyhľadávacia brána. Uvedené komponenty budú v kapitolách 3 a 4 použité pri návrhu a implementácii výsledného webového vyhľadávacieho systému. Pokiaľ niektorá komponenta, prípadne funkcionalita nebude vyhovovať vstupným požiadavkám, tak poslúži ako odrazový mostík k implementácii riešenia, ktoré kompletne napĺňa vstupné požiadavky.

Poznámka: Podrobný popis uvedených komponent siaha nad rámec tejto diplomovej práce. Práve preto si predstavíme a popíšeme ich hlavnú úlohu a základné vlastnosti. Predovšetkým z pohľadu funkcií podstatných pre návrh a implementáciu webového vyhľadávacieho systému. Príloha B tejto diplomovej práce obsahuje kompletný súhrn dostupných zdrojov venovaných popisu uvedených komponent.

2.1 Middleware J5M

Middleware J5M [11] predstavuje jednoduchý rámec zameraný predovšetkým na rýchly vývoj aplikácií v distribuovanom prostredí. Z hľadiska základnej funkcionality poskytuje v podstate rovnaké možnosti ako platforma JINI [20]. Navyše ale obsahuje niekoľko funkcií, ktoré nie sú dostupné na štandardnej JINI [20] platforme. Patrí medzi ne predovšetkým priame volanie Java kódu namiesto RMI [18, 19], vylepšené možnosti vyhľadávania záujmových objektov, asynchrónne volania či možnosť klonovania bežiacich objektov. Praktickou funkciou J5M sú predovšetkým takzvané *live* parametre. Každý vzdialený objekt má možnosť deklarovať pri štarte parametre, ktoré sú prostredníctvom kontextu podkladovej vrstvy dostupné ostatným prvkom distribuovaného systému. Navyše, s použitím *live* parametrov môže programátor obmedziť vyhľadávanie vzdialených objektov tak, že získa iba referencie na inštancie tých objektov, ktorých aktuálne hodnoty *live* parametrov sa zhodujú s podmienkami lookup dotazu.



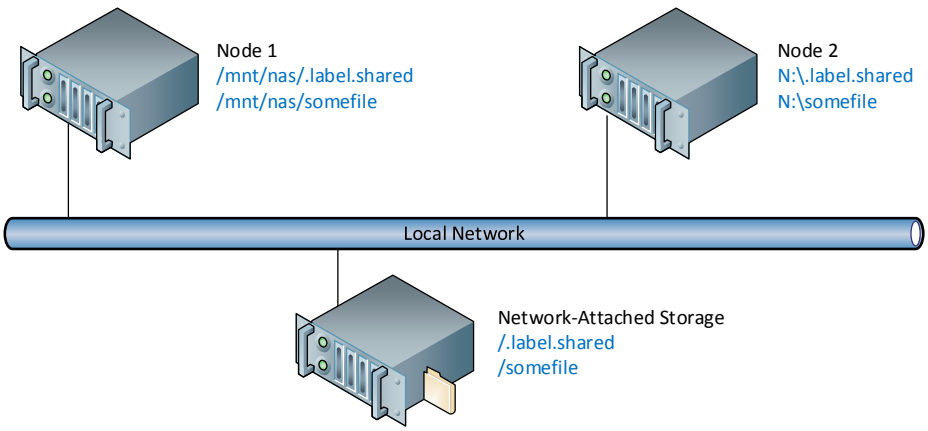
Obr. 2.1: Príklad nasadenia J5M (zdroj: J5M Programming Guide [21])

Okrázok 2.1 znázorňuje príklad nasadenia distribuovanej aplikácie implementovanej nad J5M. Základným prvkom J5M je takzvaný *nugget*, ktorý je analógiou operačného systému v kontexte bežných počítačov. *Nugget* je teda podkladovým prostredím, ktorý hosťuje inštancie takzvaných *vzdialených objektov*. Vzdialené objekty sú v podstate business objekty, ktorých rozhranie je s pomocou J5M publikované a prístupné v distribuovanom prostredí. Vzdialené objekty sú v systéme dostupné za pomoci stromovej mennej štruktúry. Doménové meno, ako napríklad `//domain1/connector/filesystem` reprezentuje objekt z domény `domain1` a lokálnym menom `/connector/filesystem`. Výhodou J5M je fakt, že v systéme môže byť nasadených hneď niekoľko objektov rovnakého mena. Aplikácia síce komunikuje vždy len s jediným konkrétnym objektom, ale v prípade jeho nedostupnosti systém presmeruje všetky požiadavky na inú / aktuálne dostupnú inštanciu. Nasadenie J5M je možné v dvoch variantách, a to:

- s použitím LDAP servera,
- alebo multicastu.

Výhodou nasadenia s použitím LDAP servera je predovšetkým možnosť distribúcie a komunikácie objektov v rámci rôznych, i verejných počítačových sietí. V takomto prípade nie je nutná úprava konfigurácie sieťových prvkov. Nasadenie s využitím sieťového multicastu je vhodné predovšetkým k použitiu v homogénnych klustroch v rámci jednej podsiete. Samotný multicast je navyše omnoho rýchlejší ako LDAP varianta.

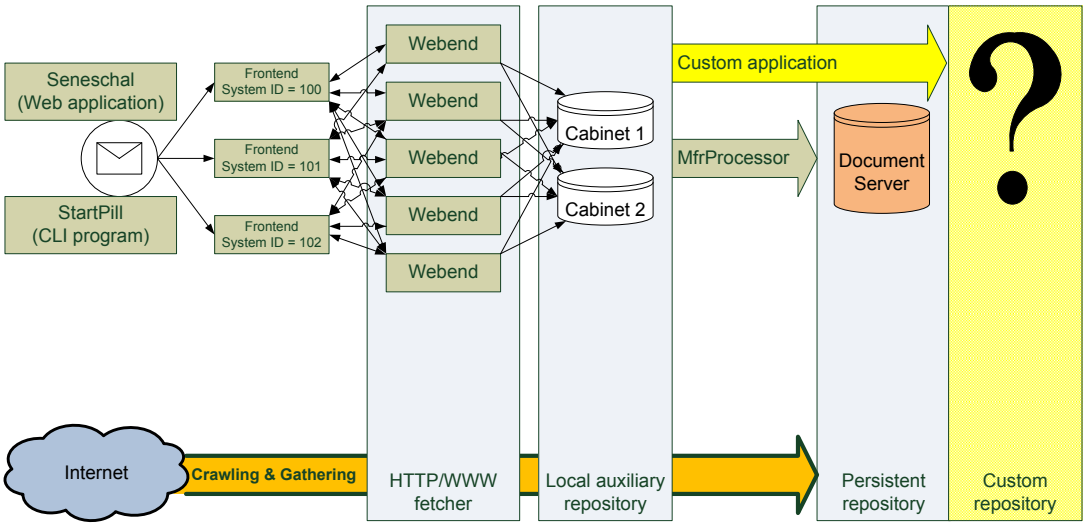
Poslednou, ale nie menej dôležitou funkciou J5M je podpora pre nasadenie zdieľaných súborových systémov. Výpočty distribuované medzi rôzne uzly môžu vyžadovať prístup k rovnakým súborom na zdieľanom úložisku. Predovšetkým v heterogénnych klustroch môže byť takéto zdieľané úložisko mapované pod rôznymi cestami. Predávanie referencií na súbory a ich prevod na cestu vrámci daného uzlu môže spôsobovať značné komplikácie. Middleware J5M poskytuje nástroj na riešenie popísaného problému. Umožňuje označiť ľubovoľný záujmový adresár na disku takzvaným *labelom*, respektíve značkou. Middleware pri štarte *nuggetu* prehľadá určenú časť súborového systému a zaznamená cesty k všetkým značkám. Programátor má potom možnosť dynamicky previesť konkrétnu značku na absolútnu cestu v rámci lokálneho súborového systém. Lokálna cesta následne poslúži ako prefix relatívnej cesty vrámci takto mapovaného adresára. Obrázok 2.2 znázorňuje zdieľané úložisko dostupné z rôznych výpočtových uzlov. Súbor `somefile` uložený na zdieľanom úložisku vidí server `Node 1` v adresári `/mnt/nas`. Naopak, `Node 2` nájde požadovaný súbor v koreňovom adresári pripojeného sieťového disku `N:\`. Vzdialené objekty na oboch uzloch si tak môžu zaslať iba relatívnu cestu k súboru `somefile`, ktorú si s použitím prefixu prevedú na absolútnu cestu v rámci ich lokálneho súborového systému.



Obr. 2.2: Príklad použitia zdieľaného súborového systému.

2.2 Webový crawler BOBO

Webový crawler BOBO tvorí univerzálnu a distribuovanú vyťažovaciu architektúru, ktorá je schopná vykonávať zadania ľubovoľného typu. Konkrétne, masívne sťahovanie webu, klasické/cielené sťahovanie webových stránok, prípadne môže fungovať ako samostatná virtuálna entita.



Obr. 2.3: Príklad nasadenia crawlera (zdroj: BOBO Manuál)

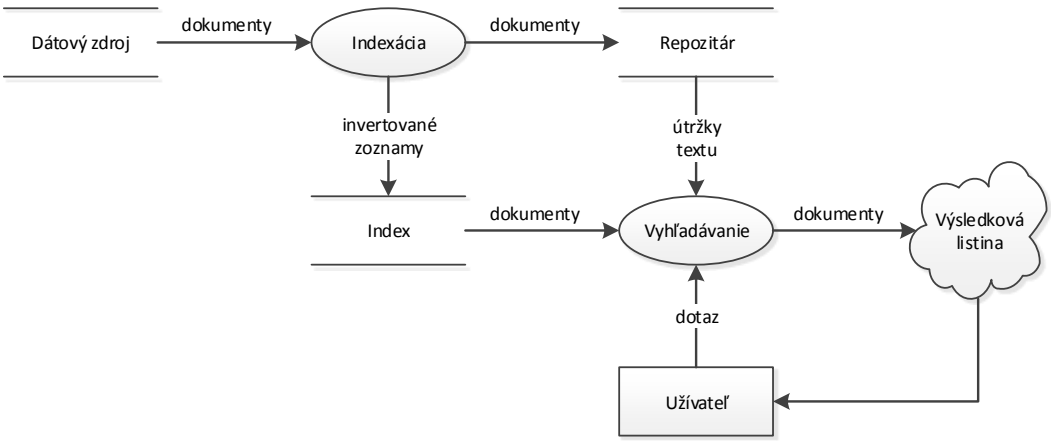
BOBO pozostáva z troch základných častí, konkrétne z *frontendu*, *webendu* a *kabinetov*. Frontend slúži ako vstupný bod pre zadávanie a registráciu jednotlivých hlavných vyťažovacích úloh, ich plánovanie a registráciu samostatných podúloh. Webend slúži ako skutočná vykonávacia rutina naplánovaných úloh. V drvivej väčšine implementácii funguje ako sťahovadlo webových stránok. Konkrétna politika sťahovania však závisí od vlastnej implementácie danej služby webendu. Posledným funkčným celkom je takzvaný kabinet. Ide o dátové úložisko, do ktorého webend odosiela / ukladá zozbierané dáta. Kabinety sú po svojom naplnení dostupné na spracovanie akýmkoľvek užívateľským procesom, ktorý je schopný prečítať ich známu štruktúru.

Celkový proces vyťažovania prebieha zhruba nasledovne. Užívateľ zadá konkrétnu úlohu v tvare, ktorý sa dá voľne popísať nasledujúcou formou: "chcem vyťažovať množinu URL adries, začínajúc ako robot menom R na adrese A, po dobu T a získané dáta chcem zapísať do kabinetu C". Po tom, čo systém začne definovanú úlohu vykonávať, je získaný obsah odosielaný do cieľového kabinetu. Pokiaľ zadaný kabinet neexistuje, tak systém získané dáta jednoducho zahodí.

Nevýhodou uvedeného konceptu kabinetov je nutnosť ich manuálneho spúšťania. Bobo v aktuálnej verzii nedisponuje možnosťou dynamického vytvárania kabinetov programovou cestou. Navyše, každý kabinet pozostáva z množstva takzvaných chunkov. Každý chunk je možné prečítať a spracovať až po jeho naplnení a uzatvorení, teda po uložení zhruba 10 000 dokumentov. Uvedené nevýhody je ale možné prekonať s vynaložením relatívne malého úsilia. Popisu takéhoto riešenia je venovaná kapitola 3.2.2.

2.3 Egothor2

Projekt Egothor2 [26] je akademickým projektom, ktorý vznikol v roku 2006 a bol výsledkom snahy o zacelenie priepasti v oblasti vyhľadávacích strojov pre akademické použitie. Primárnym cieľom bolo vytvorenie produktu, ktorý by dokázal transparentne pracovať v lokálnom ale aj distribuovanom prostredí. Navyše, existovala požiadavka aby takéto riešenie z architektonického hľadiska podporovalo rozšírenia či implementácie ľubovoľných vyhľadávacích stratégií. Výsledkom bola knižnica Egothor, ktorá implementovala fulltextový vyhľadávací stroj podporujúci predovšetkým dve základné operácie indexácie a vyhľadávania. Knižnica bola v nasledujúcich rokoch základom niekoľkých ďalších prací. Konkrétne sa práce snažili rozšíriť jadro systém o nové zaujímavé funkcie. Išlo napríklad o návrh a implementáciu algoritmov transakčného spracovania a riadenia konkurenčného prístupu k indexu [22], detekcie plagiátov [23] či nedávnu implementáciu kešovacích stratégií [24].



Obr. 2.4: Schéma dátových tokov (zdroj: Egothor2 [26])

Okrázok 2.4 ilustruje dátové toky, ktoré prebiehajú v rámci jadra vyhľadávacieho stroja. Vstupom do systému sú dokumenty určené k indexácií, konkrétne textové dáta. Egothor pri procese indexácie vytvára zo vstupných dokumentov takzvané *invertované zoznamy*, ktoré následne vkladá do indexu. Zároveň však indexačná rutina ukladá, respektíve archivuje pôvodné vstupné dáta do repozitára. Pôvodný text dokumentov je tak v procese vyhľadávania dostupný pre potreby generovania krátkych útržkov z textu dokumentu odpovedajúceho dotazu. Zadanie dotazu nad indexom vyvolá proces vyhľadávania, ktorý sekvenčne prehľadá časti indexu a vygeneruje výsledkovú listinu. Samotný projekt Egothor akceptuje na vstupe indexačného procesu iba dáta vo formáte *html* alebo *plain text*. Vkladanie akýchkoľvek iných formátov vyžaduje implementáciu vlastných aplikácií, respektíve rutín, ktoré prevedú dáta zo záujmového zdroja do interných štruktúr knižnice. Celkový koncept Egothoru ako fulltextového vyhľadávača pozostáva z troch základných pojmov: *dokumentu*, *indexu* a *repozitára*.

Dokument, reprezentuje abstrakciu nad konkrétnou informačnou položku. Položku tvorí predovšetkým neštruktúrovaná textová informácia a ďalšie meta dáta ako sú napríklad referencia na pôvodný zdroj, identifikátor, číslo revízie, dátum indexácie či iné. S ohľadom na jasnosť a jednoduchosť si predstavme konkrétny súbor na lokálnom disku, napríklad `/root/egothor.txt`. Uvedený súbor je teda textovým súborom, u ktorého máme k dispozícii nie len jeho umiestnenie na disku, čas vytvorenia, čas poslednej zmeny, ale predovšetkým samotný textový obsah tohto súboru. V rámci Egothoru potom takýto súbor pri procese indexácie a následného vyhľadávania musíme reprezentovať ako samostatný dokument. Konkrétne, knižnica Egothor k tomu poskytuje triedu `org.egothor.core.memory.Document`. Autor aplikácie, ktorá bude využívať indexačné / vyhľadávacie mechanizmy, musí pri implementácii zabezpečiť inicializáciu tohto objektu s použitím vstupných dát. V našom príklade, pokiaľ chceme indexovať súbor `egothor.txt`, tak naša aplikácia musí z uvedeného súboru získať všetky meta dáta a obsah súboru. Takto získanými hodnotami potom inicializovať inštanciu triedy `org.egothor.core.memory.Document`.

Index, je základnou dátovou štruktúrou vyhľadávacieho stroja a slúži predovšetkým na rýchle vyhodnotenie užívateľského dotazu a vrátenie zoznamu odpovedajúcich dokumentov. Index knižnice Egothor má podobu takzvaného invertovaného indexu. Invertovaný index je tvorený množinou všetkých indexovaných slov, respektíve tokenov. Ku každému takémuto token existuje takzvaný invertovaný zoznam, ktorý predstavuje zoznam dokumentov, v ktorých sa daný term nachádza. Konkrétne, každý záznam invertovaného zoznamu pozostáva z *identifikátora dokumentu*, v ktorom sa daný term nachádza a *váhy daného termu*. Indexácia dokumentu je teda proces, pri ktorom Egothor vytvorí z daného textového obsahu takzvaný invertovaný zoznam a ten následne pridá do globálneho indexu. Dátová štruktúra tvoriaca index je zastúpená abstraktnou triedou `org.egothor.dir.Tanker`, ďalej len Tanker. Tanker implementuje predovšetkým všetky základné operácie nutné k práci s indexom. Uvedená trieda ale žiadnym spôsobom nepodporuje konkurenčné prístupy k indexu. Problematikou transakcií sa zaoberala až práca [26]. Výsledkom bola implementácia transakčného spracovania z prostredia SRDB, konkrétne v podobe triedy `org.egothor.dir.TankerImplSecure`. Riadenie prístupu k indexu obstaráva `org.egothor.lock.LockServer`. Slúžiaci ako server, prostredníctvom ktorého získavajú nezávislé inštancie tankeru zámky nad požadovanými časťami indexu.

Repozitár, na rozdiel od indexu, neudržiava invertované zoznamy. Ide o bázu originálneho textu indexovaného dokumentu. Poskytuje tak prístup k pôvodnému textu dokumentu a k ďalším meta dátam asociovaným s daným dokumentom. Pôvodné textové obsahy dokumentov sú nevyhnutné predovšetkým kvôli potrebe generovať do výsledkovej listiny krátke útržky / ukážky z textu nájdeného dokumentu. Uvedené dáta sú v rámci interných štruktúr reprezentované triedou `org.egothor.core.DocumentData` a konkrétna inštancia tvorí dáta konkrétneho dokumentu. Repozitár publikuje dostupné funkcie prostredníctvom rozhrania `org.egothor.repository.DataRepository`, ktorého použitie umožní nasadenie ľubovoľnej implementácie. Interná implementácia repozitára je postavená s využitím natívnej databázy BerkeleyDB [27].

2.4 Web Search Engine

Web Search Engine [13], ďalej len WSE, je webový portál, ktorý vznikol roku 2009. Primárnym cieľom pri jeho návrhu a implementácii bolo vytvorenie uceleného užívateľského rozhrania nad službami Egothoru [26]. Výsledná aplikácia funguje ako standalone webový portál, ktorý sprístupňuje základné funkcie vyhľadávania nad množinou dokumentov dostupných v rámci lokálneho súborového úložiska. Navrhnuté riešenie ale žiadnym spôsobom neriešilo akvizičnú časť aplikácie, teda import dokumentov z rôznych dátových zdrojov. Užívateľ tak mohol vyhľadávať iba v dokumentoch na lokálnom dátovom úložisku. Navyše systém neposkytoval nástroje na správu takéhoto lokálneho úložiska. Užívateľ však mal možnosť prostredníctvom jednotného rozhrania využívať indexačné a vyhľadávacie služby podkladovej vrstvy systému. Centrálnym prvkom celého konceptu je takzvaný webový priestor. Pod uvedeným pojmom sa skrýva akási abstrakcia nad množinou dokumentov, v ktorých môže užívateľ vyhľadávať. K definícii webových priestorov užívateľ používa regulárne výrazy. Pomocou nich jednoducho vymedzí množinu URL adries dokumentov, ktoré spadajú do daného webového priestoru, respektíve indexu. Popísaným spôsobom tak môže užívateľ implicitne dosiahnuť zaradenie konkrétneho dokumentu hneď do viacerých webových priestorov súčasne, či naopak, vylúčiť požadované dokumenty z procesu indexácie. Ako sme už uviedli v popise vyššie, webový priestor reprezentuje množinu URL adries dokumentov. WSE bol schopný indexovať iba dokumenty umiestnené na lokálnom dátovom úložisku. Referencie na tieto dokumenty preto nie sú URL adresami v zmysle špecifikácie [16], teda syntaxe na vyjadrenie prístupových informácii k objektu na sieti. WSE rieši tento problém za pomoci `.htaccess` súborov, teda konfiguračných súborov webového servera Apache, rozmiestnených v rámci adresárovej štruktúry lokálneho úložiska. Konkrétne, aplikácia využíva direktívu `RedirectPermanent` k transformácii lokálnych referencií na požadovanú URL adresu.

Príklad: Nech adresár `/home/user/data/` je koreňový adresár úložiska dokumentov a nech obsahuje súbor `.htaccess` s nasledujúcou konfiguráciou:

```
RedirectPermanent /tomcat/ http://tomcat.apache.org/
RedirectPermanent /publications/ ftp://publications.company.com/
```

Ukážka 2.1: Príklad `.htaccess` súboru

Spustením procesu indexácie začne WSE prechádzať lokálnou adresárovou štruktúrou od preddefinovaného koreňového adresára. Pri prechode hľadá akékoľvek známe formáty súborov, pre ktoré má k dispozícii rutinu pre extrakciu textového obsahu. Navyše, v adresárovom strome hľadá dostupné `.htaccess` súbory tak, aby podľa nich transformoval lokálne referencie dokumentov. Napríklad, pokiaľ proces narazí na dokument `/home/user/data/tomcat/index.htm`, tak v prvom kroku získa relatívnu referenciu uvedeného súboru vzhľadom na koreňový adresár, teda `/tomcat/aio.htm`. Aplikáciou direktívy `RedirectPermanent` z ukážky 2.1 systém prevedie relatívnu cestu `/tomcat/aio.htm` na výslednú URL, ktorá bude v tomto prípade `http://tomcat.apache.org/aio.htm`. Uvedený spôsob tak do určitej mieri dovolí mapovať lokálne indexované súbory na ich on-line referencie.

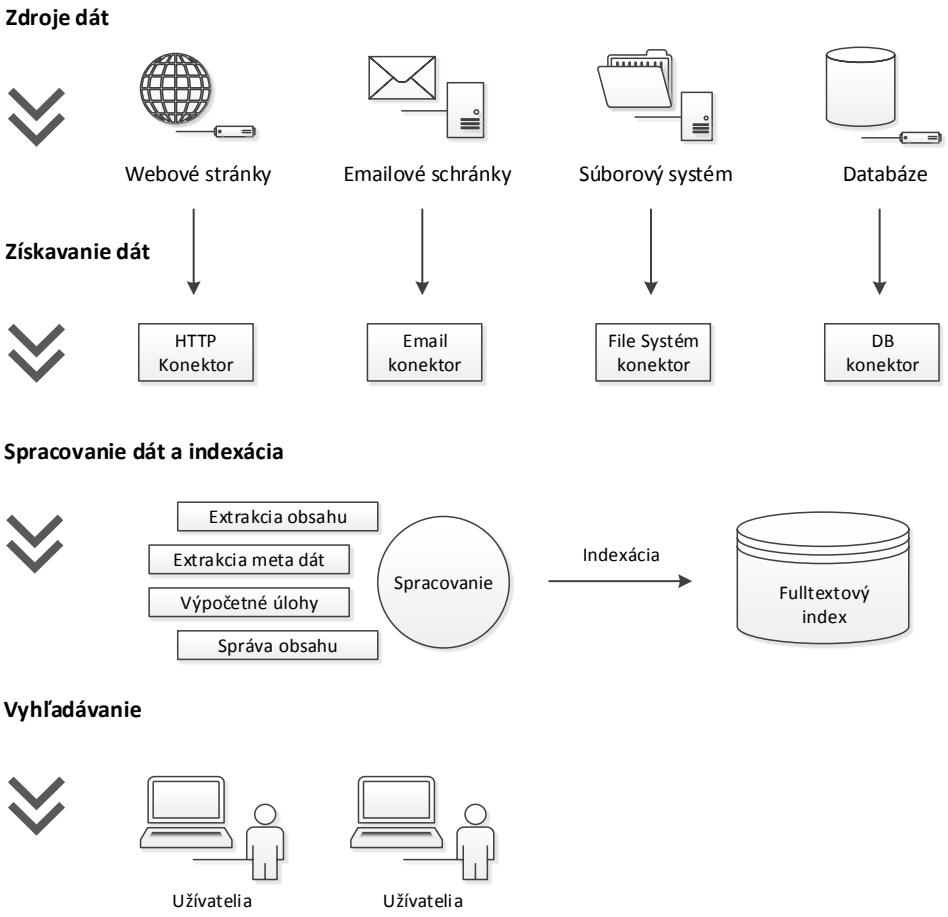
Nevýhodou, ktorá bráni komplexnejšiemu a rozsiahlejšiemu nasadeniu aplikácie WSE pre väčšie množstvo užívateľov je predovšetkým fakt, že sa jedná o standalone Java EE aplikáciu. Všetky výkonnostne náročné operácie extrakcie dát a následnej indexácie zaťažujú server, na ktorom je nasadená a aplikáciu preto nie je možné rozumne škálovať. Práve preto sa pri návrhu nového užívateľského rozhrania nášho webového vyhľadávacieho systému zameriame predovšetkým na možnosti škálovateľnosti výsledného riešenia.

Kapitola 3

Návrh

Riešenie popisované v rámci tejto práce je komplexný distribuovaný vyhľadávací systém, ktorý v sebe integruje všetky funkcie špecifikované v kapitole 1.2 a je pripravený na okamžité nasadenie. Napriek tomuto faktu sú jednotlivé časti systému rozdelené do logických celkov zameraných na vykonávanie špecifickej, presne definovanej úlohy. Z tohto dôvodu sa v prvom rade zameriame na rozdelenie celého systému do logických celkov, ktorých konkrétnym návrhom sa budeme venovať samostatne. Budeme diskutovať predovšetkým charakter prepojenia, respektíve vzájomnú komunikáciu uvedených samostatných celkov tak, aby vo výsledku splnili požiadavky na nich kladené, a to predovšetkým z pohľadu škálovateľnosti a distribuovateľnosti výsledného riešenia. Kapitola 1.1 predstavila základný koncept komplexného vyhľadávacieho systému, predovšetkým rozdelenie do primárnych celkov, teda vyťažovanie, spracovanie, extrakcia ďalších informácií, indexácia a na záver samotné vyhľadávanie. Pri návrhu jednotlivých častí systému budeme brať do úvahy ešte jeden dôležitý fakt, a to možnosť jednoduchého rozšírenia navrhnutých komponent o ďalšie funkcie. Uvedená vlastnosť navrhovaného systému síce nebola súčasťou vstupných požiadaviek, ale vzhľadom na komplexnosť a rozsah vyhľadávacieho systému je táto vlastnosť kritická predovšetkým z pohľadu budúceho rozširovania výsledného riešenia.

Pozrime sa bližšie na základné časti spoločné pre každý vyhľadávací systém. Obrázok 3.1 znázorňuje štandardné rozdelenie bežného ES systému na základné vrstvy tak, ako boli predstavené v kapitole 1.1. Navyše, uvedenou schémou sa riadia taktiež dátové toky komerčných riešení, medzi ktoré patrí aj v úvode práce spomínaný Autonomy IDOL [7] či Tovek Server [8]. Základom všetkého sú zdroje dát, nad ktorými musí systém umožniť vyhľadávanie. Systém preto musí byť v prvom rade schopný požadované dáta získať. Prvým celkom, respektíve subsystémom vyhľadávača je preto sada nástrojov, komponent, určených k získavaniu dát. Vyťažené dáta sú následne odoslané do vrstvy spracovania dát. Fáza spracovania ma za úlohu získať zo vstupov čo možno najviac *relevantných informácií* a tie uložiť do interných štruktúr systému. Konkrétny význam slovného spojenia *relevantné informácie* musíme v kontexte popisu obecného ES zámerne vynechať. Silne totiž závisí na prípade užitia vyhľadávača. Všetky takto získané či spracované informácie systém pridá do indexu tak, aby bol schopný v nich efektívne vyhľadávať. Poslednú vrstvu potom tvorí sada vyhľadávacích alebo Business Intelligence nástrojov nad rozhraním systému.



Obr. 3.1: Obecná schéma ES systému

Nesmieme však zabudnúť na skutočnosť, že vyhľadávací systém musí disponovať taktiež sadou funkcií určených k správe a riadeniu uvedených celkov. Navyše, všetky predstavené vrstvy musia byť schopné kooperovať tak, aby sa navonok pôsobili ako dokonale zosúladený celok. Interná implementácia naopak musí zamedziť príliš úzkej väzbe medzi komponentami. Dosiahneme tak kompaktne pôsobiaci systém, ktorého časti bude možné bezbolestne modifikovať či rozširovať podľa aktuálnych požiadaviek.

Zoznámili sme sa už s funkčnými celkami obecného vyhľadávacieho systému. V nasledujúcich kapitolách sa preto bližšie pozrieme na analogický návrh obdobného vyhľadávacieho systému, tak aby naplnil požiadavky stanovené v kapitole 1.2, a to predovšetkým v kontexte komponent prvej generácie systému pre spracovanie webového obsahu. Kapitola 2.1 predstavila jednu z kľúčových komponent nášho systému. Middleware J5M nám tak umožní navrhnúť a realizovať plne distribuované riešenie. BOBO bude slúžiť ako *ready-to-use* riešenie pre základný typ dátového zdroja, konkrétne obecné webové stránky. Egothor2 poslúži ako fulltextový vyhľadávací stroj nad požadovanými dátami. WSE poskytne ukážku použitia dostupných funkcií v rámci užívateľského rozhrania a bude inšpiráciou pre vytvorenie nového vyhľadávacieho portálu.

3.1 Celková architektúra

Návrh architektúry nášho webového vyhľadávacieho systému sa bude držať základného rozdelenia ilustrovaného obrázkom 3.1. Konkrétny návrh a implementácia fázy vyťažovania, spracovania a vyhľadávania sa už však u dostupných systémov líši. Základom nášho návrhu bude možnosti distribúcie. Držiac sa obecného delenia sa teraz bližšie pozrime na jednotlivé časti systému.

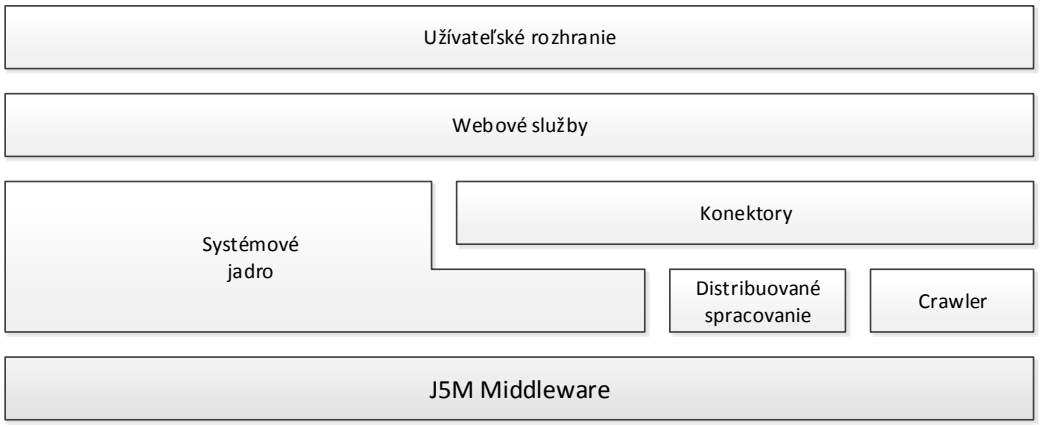
Konektory predstavujú skupinu nástrojov určených na získavanie dát zo záujmového zdroja. Pod záujmovým dátovým zdrojom si môžeme predstaviť čokoľvek od textového súboru na lokálnom súborovom systéme až po emaily na Exchange servery. Konektory prijímajú konkrétne zadania vyťažovacích úloh a na ich základe získavajú požadované dáta dostupné prostredníctvom daného dátového zdroja. Takto získané dokumenty sú priebežne odoslané do systému distribuovaného spracovania dát. Podrobný návrh systému konektorov približuje kapitola 3.2.

Komponenty distribuovaného spracovania dát predstavujú generickú platformu umožňujúcu paralelné spracovanie dát. Konkrétne majú za úlohu extrahovať zo získaných dokumentov ďalšie dáta, či meta dáta, ktoré sú významné pre systém ako celok alebo pre samotného užívateľa či administrátora systému. Vzhľadom na širokú škálu operácií, ktoré je nad dostupnými dátami možné vykonávať je dôležitým faktorom návrhu predovšetkým oddelenie rutín vykonávajúcich požadovanú úlohu od procesov, ktoré spúšťajú dané rutiny na vstupné dáta. Komponenta distribuovaného spracovania dát by mala byť implementovaná ako samostatná knižnica, ktorú by bolo možné použiť pre akékoľvek aplikácie vyžadujúce spracovanie dát. Návrhom uvedeného subsystému sa bližšie zaoberá kapitola 3.3.

Vyhľadávacie jadro systému poskytuje kompletnú funkcionalitu súvisiacu nielen s indexáciou. Jadro systému musí umožniť taktiež správu dátových zdrojov, ovládanie konektorov, správu užívateľských účtov, prístupových práv či dát samotných. Predstavuje kompletnú sadu funkcií napĺňajúcich funkčné požiadavky špecifikované v kapitole 1.2. Medzi takéto požiadavky patrí taktiež možnosť priechodu históriou získaných dokumentov. Navyše slúži ako centrálny bod systému, ktorý zjednocuje komunikáciu všetkých ostatných komponent. Zastrešuje tak správu konektorov a zadávania úloh a využíva služieb distribuovaného spracovania pre potreby spracovania a vyhľadávania dát.

Vyhľadávanie bude zastúpené webovou aplikáciou, portálom, ktoré poskytne ucelené a intuitívne užívateľské rozhranie na prácu s funkciami, ktoré sú dostupné nad podkladovou vyhľadávacou vrstvou. Každý užívateľ vyhľadávacieho systém má ale svoje špecifické požiadavky. Niektorý preferuje *all-in-one* aplikáciu, kde nájde na jednom mieste všetky požadované funkcie. Ďalším zase vyhovuje varianta, kedy je vyhľadávací systém začlenený do už existujúcich väčších informačných celkov. Užívateľské rozhranie preto navrhneme ako *all-in-one* webový portál. Bude slúžiť ako prototyp aplikácie postavenej nad službami backendových systémov. Poslúži tak ako komplexný príklad, ukážka nasadenia a použitia všetkých funkcií systému.

Obrázok 3.2 ilustruje návrh rozdelenia uvedených funkčných celkov tak, ako sme ich popísali v predošlých odstavcoch. Základom celého distribuovaného systému je middleware J5M. Nad ním bežia všetky komponenty jadra vyhľadávača, distribuovaného spracovania a konektorov. Uvedené komponenty budú komunikovať výhradne prostredníctvom J5M čím zaručíme distribuovateľnosť celého systému. Naplnenie požiadavky 7 z kapitoly 1.2 si vyžiada pridanie vrstvy webových služieb. Komponenta webových služieb publikuje všetky verejné funkcie jadra a konektorov ako štandardnú webovú službu. Nakoniec, webový portál používa výhradne iba API webových služieb. Uvedené zaručí úplnú izoláciu užívateľského rozhrania od interných implementácií distribuovaného systému.



Obr. 3.2: Koncept rozdelenia vrstiev navrhovaného systému nasadenia

3.2 Import dát do systému

Neoddeliteľnou súčasťou vyhľadávacieho systému sú v prvom rade komponenty určené k získavaniu dát. Aby bol akýkoľvek systém schopný vyhľadávať v sade dokumentov, tak musí predovšetkým vytvoriť dátovú štruktúru, takzvaný index. Ten na položený dotaz okamžite vráti zoznam dokumentov spĺňajúcich zadané podmienky. Stavba indexu vyžaduje aby mal vyhľadávací stroj v čase indexácie k dispozícií pôvodný zdroj informácie, v našom prípade teda indexovaný súbor či internetovú stránku. Úlohu získavania dát pre proces indexácie vyriešime v rámci návrhu nášho systému využitím konceptu takzvaných dátových zdrojov a konektorov.

Dátový zdroj predstavuje abstraktný pojem, pod ktorým sa skrýva v podstate akýkoľvek zdroj informácií, v ktorých má náš systém umožniť vyhľadávanie. Príkladom dátového zdroja môže byť lokálny / sieťový disk, v rámci ktorého môžeme prechádzať adresárovou štruktúrou a čítať uložené súbory. Inými typmi dátových zdrojov môžu byť napríklad Microsoft Exchange Sever, relačné databáze, diskusné fóra, chaty, sociálne siete, RSS kanály atď. Uvedené príklady názorne ilustrujú existenciu širokej škály rôznorodých typov dátových zdrojov. Navyše, rôzne aplikácie uchovávajú a sprístupňujú uložené dáta rôznym spôso-

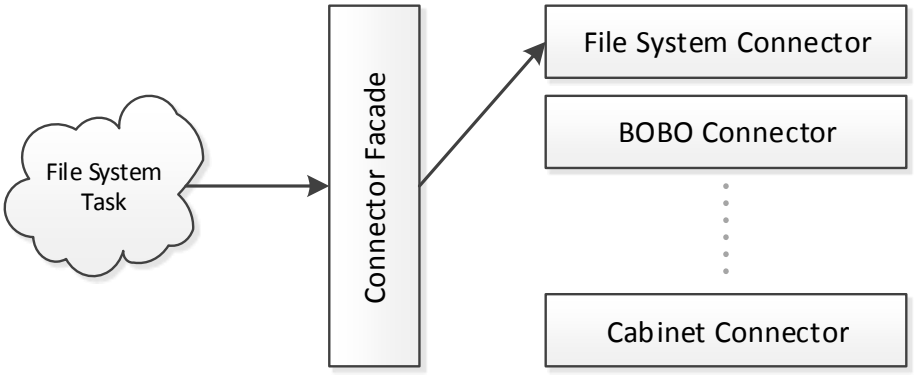
bom. Koncept dátového zdroja preto musí byť z pohľadu užívateľa, či celého systému čo možno najobecnejší. Musí byť schopný reagovať na popísanú rôznorodosť a poskytnúť mechanizmus na popis daného dátového zdroja. Ako obecný popis dátového zdroja sa ponúka skupina konfiguračných parametrov, ktoré špecifikujú minimálne nasledujúce:

- typ zdroja
- umiestnenie zdroja
- spôsob prístupu, prípadne prístupové údaje
- ďalšie parametre pre popis dátového zdroja
- sada parametrov pre proces spracovania

Uvedený zoznam vedie na použitie plne generického popisu dátového zdroja. Z obecného hľadiska sa preto ako najvhodnejší spôsob javí použitie sady konfiguračných parametrov. Konkrétne zoznamu dvojíc v tvare kľúč/hodnota. Spoločným parametrom pre akékoľvek zadanie bude parameter, ktorý určí konkrétny typ zdroja. Na základe parametra potom systém identifikuje správny typ konektora, určeného na vyťažovanie takto definovaného dátového zdroja. Konkrétna vyťažovacia rutina už bude rozumieť všetkým ďalším konfiguračným parametrom predaným vrámci takéhoto generického popisu zdroja. Navrhovaný generický popis dátového zdroja následne použijeme ako vstupné zadanie pre konektor. Dosiahneme tak zjednotenie popisu zdroja, spôsobu získavania dát a v neposlednej rade spôsobu ich spracovania. Následne bude možné takéto komplexné špecifikácie transparentne udržovať a modifikovať napríklad v rámci spoločnej databázy dátových zdrojov.

Konektor je samostatne bežiaca aplikácia, určená na vyťažovanie dát z konkrétneho dátového zdroja. Vstupom do konektora je špecifický popis dátového zdroja, teda vyššie popisovaná skupina konfiguračných parametrov. Konektor na základe znalosti konfigurácie vykonáva skupinu operácií, ktorých výsledkom je získanie dát z daného zdroja. Napríklad konektor pre súborový systém rekurzívne prechádza zadaný adresár a dáta odosiela do systému distribuovaného spracovania. Pred odoslaním na spracovanie konektor ešte doplní dáta o parametre, na základe ktorých budú dokumenty spracovávané a v závere zaradené do správnej kolekcie. Problémom zostáva otázka, ako určiť, že dané generické zadanie je určené presne danému typu konektora, respektíve ako určiť akému konektoru zadanie zaslať. Riešenie sa ponúka vo forme takzvanej konektorovej fasády, cez ktorú budú smerované všetky generické zadania vyťažovacích úloh. Ide o špeciálny typ konektora, ktorý síce poskytuje rovnaké rozhranie ako ktorýkoľvek iný konektor, ale interne funguje na úplne inom princípe. Fasáda nebude vykonávať žiadne zadanie. Naopak, jej primárnou úlohou bude vyhľadať v distribuovanom prostredí akúkoľvek inštanciu konektora schopného vyťažiť daný typ dátového zdroja. Vstupné zadanie mu potom priamo odošle. Použitie takéhoto návrhu nám umožní jednoducho implementovať a pridávať ďalšie typy konektorov pre nové dátové zdroje. Nasadenie nového konektora do systému potom nebude vyžadovať žiadne

dodatočné zásahy do zvyšku systému. Samozrejme s výnimkou užívateľského rozhrania, kde bude nutné doplniť možnosť zadania parametrov nového typu úlohy, napríklad vyplnením potrebného formulára. Konkrétny príklad použitia konketorovej fasády znázorňuje obrázok 3.3.



Obr. 3.3: Schéma zadávania vyťažovacích úloh

3.2.1 Konektor pre súborový systém

Lokálny súborový systém patrí vo svojej podstate medzi najjednoduchšie dátové zdroje. Primárnou úlohou konektora je rekurzívny priechod lokálnym súborovým systémom a odosielanie obsahu nájdených súborov na ďalšie spracovanie. Pri konfigurácii vyťažovacej úlohy si tak vystačíme s využitím jediného konfiguračného parametra, ktorým je cesta k počiatočnému adresáru.

Poznámka: Ako sme už uviedli v predošlom odstavci, konektor vyťažuje iba dáta dostupné v rámci lokálneho súborového systému, ktorý býva bežnému užívateľovi nedostupný a to hlavne dôvodu bezpečnosti. Ako teda užívateľovi umožniť indexáciu vlastných off-line dokumentov? Vhodným riešením je návrh mechanizmu, ktorý transparentným a bezpečným spôsobom dovolí užívateľovi nahrávať dáta priamo na server. Odpoveď na túto otázku však siaha nad rámec tejto kapitoly. Podrobnému rozboru a implementácii výsledného riešenia sa venuje kapitola 4.4.

3.2.2 Konektor pre webového robota

Primárnym cieľom robota je predovšetkým získavanie dát z prostredia internetu a to na základe konkrétnych požiadaviek užívateľa. Aktuálna verzia robota pritom ponúka nasledujúce dva metódy sťahovania webového obsahu. Obe uvedené metódy, respektíve typy úloh budeme z pohľadu konektora chápať ako dve samostatné dátové zdroje so špecifickou konfiguráciou, teda:

- masívne vyťažovanie internetu
- cielené vyťažovanie domény

Masívne vyťažovanie internetu je takzvaný širokopásmový zber. Predstavuje analógiu operáciám klasických webových robotov určených pre fulltextové vyhľadávače. Úloha sťahuje určitú oblasť internetu, ktorá ja obmedzená s použitím regulárnych výrazov nad URL adresami stránok. Minimálna konfigurácia dátového zdroja pre tento typ úlohy by mala zahŕňať všetky aktuálne podporované parametre zberu:

- *názov robota*
- *identifikácia robota*
- *dátum a čas spustenia*
- *doba trvania*
- *zoznam počiatočných URL adries*
- *identifikácia cieľového kabinetu*
- *akceptovateľné domény*
- *akceptovateľné URL adresy*

Robot po doručení zadania novú úlohu zaregistruje a naplánuje. Plánovač sa postará o to, aby bola daná úloha spustená v požadovanom čase a ukončená po uplynutí stanoveného časového limitu. Úloha po štarte pracuje ako klasický webový crawler. Znamená to, že stiahne webové stránky dostupné na počiatočných URL adresách. Získaný obsah prechádza a hľadá ďalšie URL adresy, ktoré by mohol robot nasledovať. V závere získaný obsah uloží do kabinetu. Získané odkazy na ďalšie stránky robot ešte odfiltruje podľa domény a URL adresy, tak aby nesťahoval nežiadúce dáta.

Cielené vyťažovanie domény simuluje pohyb skutočného užívateľa na vybraných webových stránkach. Primárnym zameraním je pomalé a nenápadné stiahnutie stránok vrámci zvolenej domény a to bez aplikácie akýchkoľvek obmedzení zo strany `robots.txt`. Minimálna konfigurácia dátového zdroja pre tento typ úlohy by mala zahŕňať všetky aktuálne podporované parametre zberu:

- *identifikácia robota*
- *počiatočná URL adresa*
- *identifikácia cieľového kabinetu*
- *maximálny počet kliknutí*
- *interval medzi kliknutiami*
- *maximálna dĺžka trvania zberu*
- *interval, po ktorom sa zber opakuje*

Robot rovnako, ako u predošlého typu úlohy, po doručení zadania novú úlohu zaregistruje a naplánuje. Narozdiel od masívneho sťahovania, daná úloha prechádza postupne iba stránkami na rovnakej doméne akú má vstupná URL.

Výstupom oboch typov úloh sú takzvané kabinety predstavené v kapitole 2.2. Stiahnuté dáta, uložené do korektne ukončených kabinetov, sú tak k dispozícii na ďalšie spracovanie.

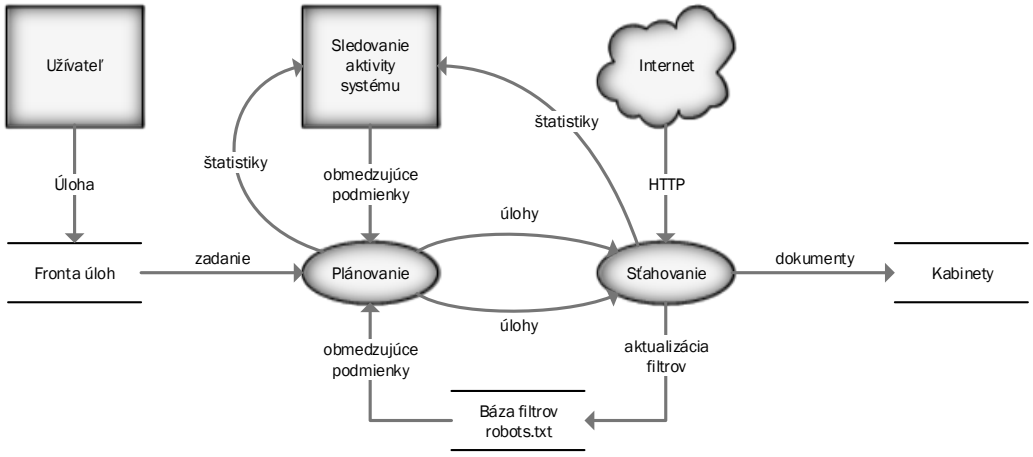
Indexácia kabinetov je síce funkcia postavená mimo robota, ale z pohľadu konektora ide o operáciu, ktorú musí konektor vykonať vždy po dokončení ktorejkoľvek vyťažovacej úlohy. V zmysle vnímania dátových zdrojov, tak ako boli predstavené v predošlej kapitole, sa nám ponúka možnosť považovať kabinet za ďalší, tretí typ dátového zdroja. Takýto typ dátového zdroja potom z hľadiska návrhu a implementácie predstavuje modifikáciu operácie vyťažovania dát z lokálneho file systému. Navyše, skrytie indexácie kabinetu pod dátový zdroj nám poskytne hneď dve nezanedbateľné výhody:

1. Poskytne nám možnosť opätovnej indexácie archívnych dát, starších zberov, ktoré by sme potrebovali znova indexovať, napríklad zo starých záloh.

2. Umožní nám separovať a zjednotiť rutinu indexácie práve stiahnutých dát po ukončení vyťažovacej úlohy. Implementácie oboch dátových zdrojov, teda ako pre masívne tak cielené zbery budú môcť získané kabinety indexovať tak, že po ukončení úlohy jednoducho interne vytvoria novú úlohu indexácie výsledného kabinetu a odošlú ju konektoru. U prijatej úlohy už konektor nerozlišuje fakt, či bola zadaná užívateľom alebo konektorom. Konektor tak splní zadanie tým, že webové stránky uložené v danom kabinete postupne odosiela k indexácii.

Konektor pre robota BOBO je z hľadiska návrhu značne komplikovanejší než predošlý konektor. Získavanie dát zo súborového systému prebieha v podstate v reálnom čase, pretože dátovým zdrojom je konkrétny adresár na disku. Dátové toky v rámci systému sú ale omnoho rozsiahlejšie, čo ilustruje aj schéma na obrázku 3.4.

Odoslaním zadania konkrétneho vyťažovania ešte nezískame okamžitý prístup k požadovaným dátam. V okamihu zadania úlohy užívateľom, či v našom prípade konektorom, vyťažovacia platforma danú úlohou v prvom rade zaradí do fronty úloh. Tam čaká až do doby, kým si ju nevyzdvihne plánovač, ktorý pre ňu následne registruje jednotlivé čiastkové úlohy. Až po uvedených krokoch začne konkrétna vyťažovacia rutina so sťahovaním dát priamo z internetu. Získané dáta sú priebežne ukladané do cieľového kabinetu. Primárnym konfiguračným parametrom kabinetu je jeho meno a absolútna cesta k adresáru, kde bude dáta zapisovať vo forme takzvaných chunkov. Každé zadanie vyťažovacej úlohy preto musí obsahovať názov vybraného kabinetu. Na základe tohto názvu potom vyťažovacie rutiny vedia, do ktorého kabinetu odosielať stiahnuté dáta. Práve kabinety sú problematickým prvkom pri návrhu automatizovaného konektora. Každý kabinet je v rámci distribuovaného prostredia vlastne samostatnou inštanciu vzdialeného objektu a musí byť v požadovanej konfigurácii inicializovaný už pri štarte podkladovej vrstvy, takzvaného *nuggetu*, viď kapitola 2.1. Inými slovami, už pri štarte



Obr. 3.4: Schéma dátových tokov v rámci platformy BOBO

vyťažovacej platformy musíme dopredu spustiť kabinety s požadovaným názvom tak, aby do nich neskôr po zadaní úlohy mohli vyťažovacie rutiny ukladať dáta. V tomto bode však narážame problém. Každý dátový zdroj, teda konkrétna oblasť internetu, ktorú chceme v rámci konektora vyťažiť, predstavuje samostatnú skupinu dát. Z tohto dôvodu musíme každému dátovému zdroju priradiť vlastný kabinet tak, aby bola zaručená separácia dát medzi zdrojmi a v neposlednej rade medzi užívateľmi. Navyše, každý užívateľ si môže počas celého životného cyklu systému vytvoriť v podstate neobmedzené množstvo vlastných dátových zdrojov. Neexistuje preto žiadna možnosť ako dopredu zistiť, aké kabinety a s akou konfiguráciou spustiť pri prvotnom štarte vyťažovacej platformy.

Popísaný problém má na prvý pohľad priamočiare riešenie. Stačí, aby systém naštartoval nový kabinet so špecifickou konfiguráciou až vo chvíli, kedy dôjde k spusteniu, respektíve štartu vyťažovania dátového zdroja. Uvedomením si hlbších súvislostí sa ale dostávame až k middlewareu J5M, od ktorého tak vyžadujeme možnosť nasadenia nového vzdialeného objektu, v našom prípade kabinetu, na aktívny *nugget*. Bohužiaľ, pôvodná verzia J5M dostupná na začiatku riešenia tejto diplomovej práce túto funkcionalitu neposkytovala. Po dohode s autorom projektu však došlo k špecifikácii presných požiadaviek a nakoniec k finálnej implementácii požadovanej funkcie. Podkladová vrstva tak síce s ohľadom na bezpečnosť nebola rozšírená o ad-hoc deployment akéhokoľvek nového objektu, ale pribudla schopnosť klonovania už existujúcich inštancií. Výsledná implementácia tak plne postačuje na naplnenie požiadavky dynamického deploymentu nových inštancií. Konkrétne, pri štarte vyťažovacej platformy tak stačí spustiť iba prototyp kabinetu. Po spustení vyťažovania konkrétneho dátového zdroja tak systém naklonuje dostupný prototyp s použitím novej konfigurácie.

Aby vrámci systému nedochádzalo k nekontrolovateľnému nárastu počtu aktívnych kabinetov a teda zbytočnému plytvaniu systémových prostriedkov, je nutné kabinety po dokončení vyťažovania uzatvárať. Ukončenie úlohy, respektíve požiadavku na ukončenie úlohy vieme odvodiť priamo z informácií dostupných

vo fronte úloh. Vyťažovacia platforma BOBO ale funguje v podstate ako vlak. Prijatím úlohy sa spustí proces, ktorému chvíľu trvá kým sa rozbehne a začne naplno vyťažovať. Rovnaké pravidlo platí pre ukončenie vyťažovania. Interval medzi prijatím požiadavky na ukončenie vyťažovania a skutočným zastavením sa pohybuje rádovo v jednotkách až desiatkach minút. Ďalším dôležitým faktom je skutočnosť, že kabinet je izolovanou komponentov, ktorá nemá žiadne väzby na ostatné časti systému a nemá tak možnosť dozvedieť sa o ukončení úlohy, ktorá do neho dáta odosielala. Potrebu automatického monitoringu stavu konkrétnej úlohy a ukončovania kabinetu preto naplníme návrhom proxy objektu pre štandardný kabinet. Proxy kabinet bude vzdialeným objektom, ktorý popri štandardnom rozhraní zastupovaného kabinetu, bude navyše implementovať serverovú metódu monitorujúcu stav priebehu vyťažovania. Metóda bude v pravidelných časových intervaloch kontrolovať, či je zadaná úloha stále aktívna. V opačnom prípade sa proxy prepne do takzvaného *shutdown* módu, v ktorom pred skutočným ukončením zotrvá ešte určitú dobu. Po uplynutí stanoveného intervalu dôjde k ukončeniu zastupovaného kabinetu a odoslaniu interného požiadavku na indexáciu kabinetu späť na konektor.

Na záver kapitoly teda ešte raz stručne zhrnieme návrh konektora pre vyťažovaciu platformu. Konektor bude podporovať vyťažovanie a indexáciu dát, respektíve dokumentov z troch typov dátových zdrojov - obecné webové stránky, konkrétna doména alebo kabinet. Kabinet obsahuje dáta uložené ostatnými typmi dátových zdrojov na lokálnom súborovom systéme, ku ktorému môže konektor okamžite pristúpiť a sekvenčným priechodom odoslať všetky dáta na indexáciu, respektíve do komponent distribuovaného spracovania. Zvyšné dva typy dátových zdrojov fungujú tak, že po štarte, v prvom kroku spustia nový kabinet o požadovanej konfigurácii a odošlú zadanie úlohy priamo vyťažovacej platforme. Po skončení vyťažovania na strane robota, konektor uzavrie daný kabinet a sám sebe odošle internú požiadavku na indexáciu práve uzatvoreného kabinetu. Implementáciou popisovaného návrhu bude naplnená požiadavka 1 z kapitoly 1.2.

3.3 Distribuované spracovanie

Primárna úloha, ktorá predchádza vyhľadávaniu je indexácia dokumentov. Vstupom indexácie sú však výhradne textové dáta, prípadne ďalšie meta dáta, tak ako sme ich popísali v kapitole 2.3. Avšak, vstupné dáta prichádzajúce z konektorov môžu reprezentovať informácie uložené v rôznych binárnych či textových formátoch a preto musíme tieto dáta najprv spracovať. Samotný proces spracovania vstupných dát musíme predovšetkým paralelizovať tak, aby sme boli schopní rozložiť vznikajúce zaťaženie medzi jednotlivé prvky distribuovanej architektúry. Spracovanie vstupných dát patrí medzi obecné problémy, ktoré nemusia súvisieť len s procesom indexácie dát. Návrh komponenty distribuovaného spracovania dát preto pojmeme ako obecnú knižnicu. Implementáciou konkrétneho riešenia nad týmto rámcom potom bude možné napĺňať konkrétne spracovateľské úlohy.

Komponenta spracovania dát bude pozostávať zo štyroch základných prvkov, ktorých podrobnému popisu sa budeme venovať v ďalších kapitolách. Zaveďme v prvom rade konkrétne základné pojmy:

- *processable* - entita, nositeľ informácie, ktorú chceme spracovať
- *dataset* - perzistentná množina entít typu *processable*
- *processor* - rutina pracujúca s entitami typu *processable*
- *worker* - aplikácia, ktorá na entity *processable* z daného *datasetu* spustí sekvenciu *processorov*

Uvedené prvky, respektíve pojmy teraz využijeme k definícii obecného rámca pre distribuované spracovanie dát. Prvým pojmom *processable* reprezentujeme v našom návrhu akýkoľvek objekt, entitu ktorá predstavuje konkrétnu skupinu informácií. Ako príklad môžme použiť súbor na disku. Entitu *processable* potom tvorí napríklad Java objekt obsahujúci originálne dáta súboru, referenciu a ďalšie meta dáta. *Dataset* funguje ako perzistentné úložisko, v ktorom sú uložené *processable* objekty. Každý takýto *dataset* je pomenovaný konkrétnym menom. *Dataset* a *processable* tvoria v našom systéme dátovú časť. Naopak *worker* a *processor* tvoria exekučnú časť. *Processor* v tomto zmysle chápeme ako rutinu, ktorá nad vstupnými dátami vykoná požadovanú operáciu v podstate ľubovoľného charakteru. Exekučným prostredím procesorov je *worker*. Úlohou *workera* je postupne získavať *processable* objekty zo vstupného *dataset*, aplikovať na nich rutiny vybraných *processorov* a takto spracované objekty zapísať do výstupného *datasetu*. Voľba vstupného / výstupného *datasetu*, ako aj sekvencia *processorov*, ktoré sa budu aplikovať na dátové objekty by mala byť súčasťou konfiguračných parametrov *workera*. Hlavnou myšlienkou je možnosť poskytnúť jednoducho rozšíriteľný model spracovania vstupných informácií. Užívateľ tak sám rozhodne, aké operácie chce nad dostupnými dátami vykonávať, a ako ich distribuovať medzi jednotlivé inštancie distribuovaného systému tak, aby vo výsledku dosiahol čo najväčšiu priepustnosť. Základnou vlastnosťou je predovšetkým podpora paralelného prístupu viacerých inštancií *workera* ku konkrétnemu *datasetu*. Konkrétna implementácia distribuovaného spracovania dát pre potreby indexácie napĺňa požiadavku 2 na navrhovaný webový vyhľadávací systém.

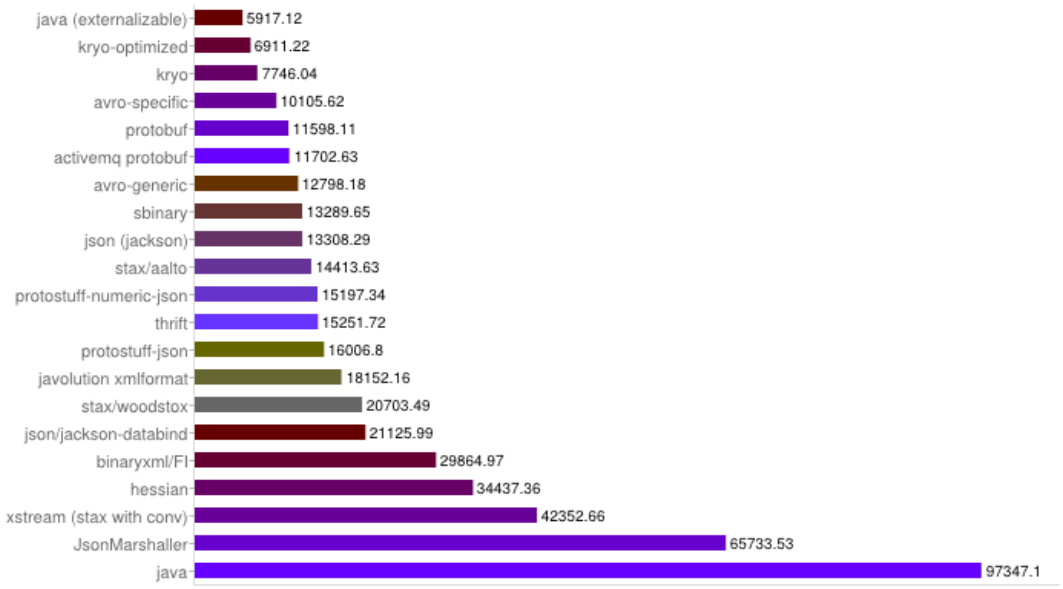
3.3.1 Processable

Processable predstavuje dátový objekt reprezentujúci zdrojový dokument, respektíve spracovávaný objekt ako celok. S ohľadom na zachovanie generickosti uvedeného dátového zdroja navrhujeme, aby jeho základné rozhranie poskytovalo prístup k nasledujúcej množine dátových položiek:

- originálne dáta
- ďalšie atribúty v tvare kľúč / hodnota

Originálne dáta predstavujú základnú množinu informácií, s ktorou jednotlivé konektory pracujú, pretože sú nositeľom primárneho obsahu spracovávaného dokumentu. Zvyšné množiny parametrov slúžia na uchovávanie dodatočných informácií o danom dokumente. Zdrojom týchto dodatočných parametrov môže byť jak samotný konektor, ktorý ich extrahuje z dátového zdroja, tak aj konkrétny procesor, ktorý extrahuje a uloží novú informáciu z pôvodných dát. Príkladom

uvedeného môže byť povedzme parameter `date.created`, ktorý získa a uloží FileSysem konektor pri načítaní súboru zo súborového systému, prípadne parameter `content.type`, ktorý doplní procesor určený na detekciu typu formátu spracovávaného dokumentu. Množinu týchto informácii však obecne nie je možné dopredu špecifikovať, a to predovšetkým z dôvodu, že *Processable* reprezentuje akýkoľvek dokument, respektíve kus informácie. Navyše, implementácia akéhokoľvek nového konektora, či procesora môže pridať ďalšiu množinu nových parametrov. Vzhľadom na túto skutočnosť sa ako najvhodnejšie riešenie javí použitie párov kľuč / hodnota, pomocou ktorých vieme reprezentovať akýkoľvek parameter.



Obr. 3.5: Porovnanie rýchlosti (de)serializácie s použitím rôznych metód [40]

Dôležitým faktorom pri implementácii tohto dátového objektu, nosiča informácií, je optimalizácia procesu serializácie daného objektu. Obrázok 3.5 znázorňuje výsledky testov procesu (de)serializácie s použitím rôznych, aktuálne dostupných techník. Graf zobrazuje celkový čas potrebný na deserializáciu objektu, vytvorenie novej inštancie daného objektu a jeho serializáciu. Z grafu je zrejmé, že najnevhodnejšou metódou na implementáciu je použitie štandardného rozhrania `java.io.Serializable`. Objekty, ktoré implementujú toto rozhranie sú (de)serializované z použitím štandardného procesu využívajúceho Java Reflection API. Naopak, najrýchlejšou metódou je implementácia rozhrania `java.io.Externalizable`. Použitie uvedeného rozhrania dáva programátorovi plnú kontrolu na procesom (de)serializácie a preto navrhujeme, aby všetky objekty vyžadujúce (de)serializáciu implementovali uvedené rozhranie.

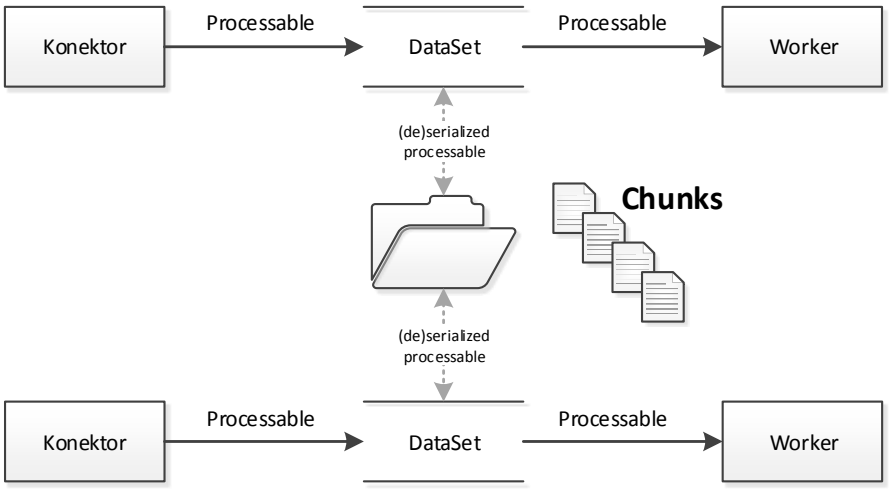
Poznámka: (De)serializácia dát prebieha nielen pri pri ukladaní a načítaní objektov v rámci datasetu, ale taktiež pri prenose dát medzi jednotlivými vzdialenými objektami v kontexte J5M. Znamená to teda, že čím optimálnejší proces implementujeme, tým rýchlejšie bude prebiehať prenos dát vo vnútri celého systému.

3.3.2 DataSet

DataSet tvorí perzistentné dátové úložisko pre ukladanie *Processable* objektov navrhnutých v kapitole 3.3.1. Základné rozhranie by preto malo obsahovať predovšetkým metódy

- **put**: metóda vloží množinu *Processable* objektov do *DataSet*u.
- **take**: metóda vráti požadovaný počet *Processable* objektov z *DataSet*u.

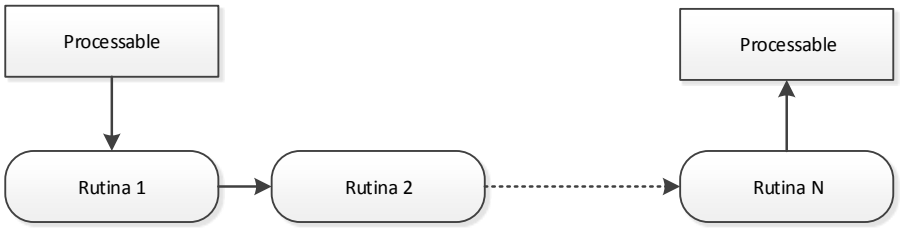
Nutnou vlastnosťou je taktiež plná podpora konkurenčného prístupu k *DataSet*u. V ľubovoľnom okamihu môže s *datasetom* pracovať hneď niekoľko iných procesov, respektíve *workerov*, a to bez ohľadu na fakt, či ide o zápis alebo čítanie *processable* objektov. Navyše, každá inštancia bude pracovať nad konkrétnym perzistentným úložiskom, kde bude uchovávať dáta podobne ako kabinety, teda v takzvaných chunkoch. Nad takýmto adresárom ale nemusí nutne pracovať jediná inštancia *datasetu*. Naopak, môže ich byť hneď niekoľko, podobne, ako ilustruje obrázok 3.6. Pri implementácii bude preto potrebné brať na zreteľ aj konkurenčný prístup k adresáru, respektíve chunkom konkrétneho perzistentného úložiska. Uvedené podmienky sa na prvý pohľad môžu zdať príliš prísne, ale implementácia, ktorá ich bude reflektovať má jednu nezanedbateľnú prednosť. Konkrétne, možnosť práce viacerých inštancií *datasetu* nad jediným spoločným úložiskom. Predstavme si, že máme zdieľané dátové úložisko, na ktorom adresár `/array/dataset` predstavuje perzistentné úložisko konkrétneho *datasetu*. Nech existuje niekoľko serverov pripojených k dátovému úložisku a nech na každom z nich beží inštancia *datasetu* pristupujúceho k zložke. V prípade nadmerného zaťaženia *datasetu* na niektorom zo serverov, má proces, ktorý s ním komunikuje, možnosť presmerovať komunikáciu na inú inštanciu *datasetu*. Zabezpečí sa tak rýchly zápis/čítanie dát do/z cieľového úložiska. Každý *dataset* by mal taktiež obsahovať vlastnú in-memory keš, aby tak eliminoval množstvo I/O operácií nad perzistentným úložiskom v prípade vysokej frekvencie operácie put a take.



Obr. 3.6: Príklad konkurenčného prístupu dvoch *DataSet*ov k zdieľanému dátovému adresáru

3.3.3 Processor

Rutinu, ktorá v nami navrhovanom systéme implementuje ľubovoľnú požadovanú operáciu nad vstupnými dátami budeme nazývať *Processor*. Každý proces by mal implementovať konkrétnu jednoduchú operáciu nad vstupnými dátami. Dosiahnutia zložitejších operácií docielime využitím sekvencie za sebou idúcich *Processorov*, ako to znázorňuje obrázok 3.7.



Obr. 3.7: Obecná sekvencia procesorov

Napríklad pre potreby indexácie použijeme sadu rutín, ktoré by vykonávali minimálne nasledujúcu množinu operácií:

- detekcia formátu vstupného dokumentu
- výpočet kontrolného súčtu nad vstupnými dátami
- extrakcia textového obsahu vstupného dokumentu
- detekcia znakovej sady extrahovaného textu
- extrakcia názvu dokumentu
- prípadná extrakcia ďalších doplňujúcich informácii
- uloženie originálneho a textového obsahu do repozitáru
- invertizácia textového obsahu
- indexácia, teda pripojenie invertovaného zoznamu do indexu
- extrakcia odkazov a uloženie do databázy

Uvedené operácie súvisia predovšetkým s extrakciou dát zo vstupných dokumentov. Procesor vo svojej podstate môže spĺňať akúkoľvek funkciu a nemusí ju ani nutne sám implementovať. Príkladom variability uvedeného konceptu je použitie procesora ako mosta pre pripojenie ďalších komponent tretích strán. Každá z uvedených operácii môže byť implementovaná ako samostatný *Processor*, ktorý rozšíri množinu informácií *Processable* objektu o nové záznamy. Ďalší procesor v sekvencii už môže s týmito novými informáciami pracovať. Napríklad správna extrakcia textu z originálnych dát vyžaduje znalosť formátu tohto dokumentu. Procesor pre extrakciu textu k tomu využíva informácie, ktoré doplnil procesor pre detekciu formátu vstupného dokumentu.

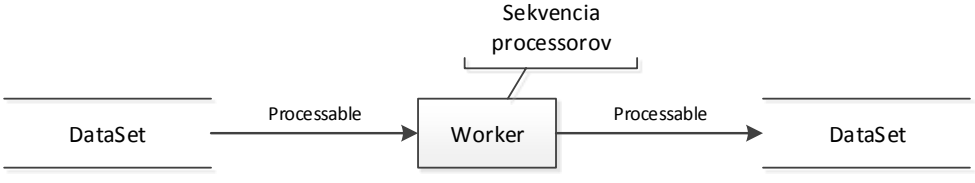
3.3.4 Worker

Doposiaľ sme si už definovali význam a primárnu úlohu dátových nosičov typu *Processable*, dátového úložiska typu *DataSet* a v neposlednej rade vykonávateľa extrakčných rutín, *Processor*. V tejto kapitole sa zameriame na posledný prvok systému distribuovaného spracovania dát, nazývaný *Worker*. *Worker* zlučuje do jednotného celku všetky doposiaľ predstavené prvky. Funguje ako aktívny prvok celého systému spracovania dát. Získava totiž *Processable* objekty priamo zo vstupného *DataSetu*. Následne ich predá na spracovanie do sekvencie *Processorov* a výsledné objekty typu *Processable* zapíše do cieľového *DataSet*. Obrázok 3.8 ilustruje najjednoduchší príklad nasadenia *Workera* a *DataSet*u. Z architektonického a výkonnostného hľadiska sa v praxi odporúča také nasadenie, pri ktorom nasadíme niekoľko *DataSet*ov a *Worker*ov za sebou.

Životný cyklus *workera* pozostáva z nasledujúcich stavov:

1. **startup**: Proces registrácie a inicializácie *Processor*ov.
2. **waiting**: Čakanie na vstupné dáta, konkrétne na neprázdny *DataSet*.
3. **loading**: Načítanie *Processable* objektov zo vstupného *DataSet*u.
4. **processing**: Aplikácia registrovaných *Processor*ov na vstupné dáta.
5. **storing**: Uloženie spracovaných dát do cieľového *DataSet*u a prechod na krok 3, kým je zdrojový *DataSet* neprázdny, inak prechod na krok 6.
6. **completition**: Finalizácia po dokončení spracovania kontinuálnej dávky a prechod do fáze 2.
7. **shutdown**: Ukončenie všetkých aktivít

Z uvedeného životného cyklu vyplýva, že základnými konfiguračnými parametrami *Worker*a je zoznam procesorov, ktoré má aplikovať na dáta a názvy vstupného a výstupného *DataSet*u. Popis životného cyklu obsahuje tiež fázu s názvom *completition*. Primárnym cieľom fáze je poskytnúť možnosť finalizácie nad spracovanou dávkou dát, konkrétne to napríklad v prípade procesora implementujúceho indexáciu dokumentov znamená vyvolanie operácie *commit* nad príslušnou inštanciou indexu. Obecne ide o vykonanie sady operácií v prípade, že dôjde k prerušeniu kontinuálneho prísunu dát pre *Worker*.



Obr. 3.8: Schéma dátových tokov pri nasadení *Workera*

3.4 Vyhľadávacie jadro

Vyhľadávacie jadro systému je centrálna časť celého navrhovaného systému a musí obsahovať všetky prvky nevyhnutné k naplneniu požiadavok z kapitoly 1.2. Problémy, ktoré musíme pri návrhu zohľadniť sa týkajú nasledujúcich otázok:

1. Ako definovať množinu dokumentov, v ktorej môže užívateľ vyhľadávať?
2. Ako bude prebiehať indexácia dokumentov?
3. Ako pracovať s vyhľadávaním?
4. Ako spravovať užívateľské práva?

Základom navrhovaného webového vyhľadávacieho systému sú dokumenty. Bez nich, respektíve bez zdroja dokumentov nie je v čom vyhľadávať. Zamyslíme sa preto nad otázkou, ako s dokumentami pracovať. Kapitola 2.4 predstavila koncept takzvaných *webových priestorov*, teda množín dokumentov, nad ktorým môžme vyhľadávať. Množina dokumentov patriacich do *webového priestoru* sa v tomto prípade definovala prostredníctvom regulárnych výrazov nad referenciami dokumentov. Uvedený spôsob plne postačuje potrebám projektu WSE, pretože ten vyhľadáva výlučne v dokumentoch z lokálneho súborového systému, kde rozmanitosť použitých referencií nie je príliš veľká. V prípade webového vyhľadávacieho systému však môžu dokumenty pochádzať v podstate z neobmedzeného množstva dátových zdrojov a rozmanitosť referencií by pri použití webových priestorov nútila užívateľa ku konštrukcii komplikovaných a nezrozumiteľných regulárnych výrazov. Navyše predpokladáme, že drvivá väčšina bežných užívateľov ani nikdy neprišla do kontaktu s regulárnymi výrazmi. Pre nich by tak definícia webového priestoru predstavovala neprekonateľný problém. Elegantné riešenie uvedeného problému sme nepriamo navrhli už v kapitole 3.2. Dátový zdroj sám predstavuje jasne definovanú množinu dokumentov, ktoré chceme zo zdroja získať. Užívateľ chce ale vo väčšine prípadov vyhľadávať nad niekoľkými zdrojmi súčasne. Posunieme sa preto ešte o úroveň vyššie a zavedieme pojem *kolekcia*. Každá kolekcia tak pozostáva z 0..N dátových zdrojov a poskytuje operáciu vyhľadávania nad množinu dokumentov vyťažených z týchto dátových zdrojov. Z technického hľadiska je potom s každou kolekciou asociovaný konkrétny fyzický index, teda tanker v kontexte projektu Egothor2. Implementáciou distribuovaného objektu, ktorého rozhranie poskytne sadu funkcií pre správu takýchto kolekcií tak naplníme požiadavku 3 z kapitoly 1.2.

Aktuálne už máme definovanú kolekciu dokumentov. Nasledovným krokom bude návrh spôsobu spracovania a indexácie dokumentov dostupných v rámci kolekcie. Kapitola 3.3 ponúka riešenie vo forme generického rámca, ktorý na vstupe získa dokument a následne ho spracuje aplikáciou sekvencie operácií implementovaných vo forme procesora. Proces indexácie tak bude vyzerať nasledovne. Konektor po spustení vyťažuje dáta zo zdrojov. Každým získaným dokumentom, napríklad webovou stránkou, inicializuje novú inštanciu *processable* objektu spolu so základnými informáciami o kolekcii. Takto získané *processable* objekty potom v dávkach odosiela na *dataset* obsahujúci doposiaľ nespracované dokumenty. V rámci jadra systému bude nasadený taktiež *worker*, ktorého úlohou bude

monitoring spomínaného *dataset*u a spracovanie dokumentov v ňom uložených aplikáciou presne definovaných *processorov*. Spracované, respektíve indexované dokumenty systém uloží do iného *datasetu* určeného k archivácií získaných dát, prípadne ich zahodí. Indexáciu dát budú teda vykonávať spomínané *processory*. Konkrétne pre potreby indexácie bude nutné postupne vykonávať minimálne detekciu formátu dokumentu, extrakciu textového obsahu dokumentu, uloženie obsahu do archívu, invertizácia a následná indexácia. Implementáciou popísaného mechanizmu distribuovaného spracovania bude naplnená požiadavka 2 z kapitoly 1.2.

Dátové zdroje, predovšetkým tie on-line, sú dynamicky sa meniacim prostredím, v ktorom každú chvíľu dochádza nielen k zmenám obsahu či zániku existujúcich dokumentov, ale taktiež k vzniku nových. Aby bol navrhovaný webový vyhľadávací systém schopný sprístupniť užívateľovi náhľad na nájdené dokumenty aj v prípade, že boli zo zdroja odstránené, musí obsahovať úložisko pre archiváciu indexovaných dát a to vrátane rôznych revízii toho istého dokumentu. Navyše, archív bude pre každý dokument udržiavať kompletnú históriu zmien, teda revízií. Archivácia originálneho aj extrahovaného textového obsahu nám tak popri náhľadu na textové dáta dáva ešte ďalšiu významnú možnosť, rekonštrukciu pôvodného obsahu. Navyše, prístupom k pôvodnému obsahu webových stránok získame možnosť transformovať originálne referencie / odkazy na interné referencie dokumentov aktuálne dostupných v archíve. Výsledkom bude lokálny off-line náhľad na indexovanú webovú stránku, z ktorej budú viesť referencie na ďalšie lokálne off-line náhľady indexovaných dokumentov. Popisované riešenie bude musieť obsahovať mechanizmus prekladu URL adresy dokumentu na identifikátor off-line verzie dostupnej v lokálnom archíve. Implementáciou uvedeného budú naplnené požiadavky 5 a 6 uvedená v kapitole 1.2.

Proces vyhľadávania zahŕňa nielen pokladanie konkrétnych dotazov, ale taktiež filtráciu podľa zvolených parametrov, akými sú napríklad formát dokumentu, veľkosť dokumentu, či dátový zdroj, z ktorého dokument pochádza. Všetky typy parametrov, podľa ktorých bude môcť užívateľ filtrovať musia byť extrahované a následne indexované aplikáciou potrebných `Processor`ov. Výstupom užívateľských dotazov, vrátane aplikácie filtračných podmienok je výsledková listina, kde každá položka bude pozostávať minimálne z názvu, referencie a krátkeho útržku z textu nájdeného dokumentu. Navyše, u každého dokumentu bude k dispozícií detailný náhľad na verziu dokumentu uloženú do repozitára pri procese indexácie. Užívateľ tak bude mať možnosť zobraziť si extrahovaný textový obsah, ako aj pôvodné dáta tak, ako sme popísali v predošlom odseku.

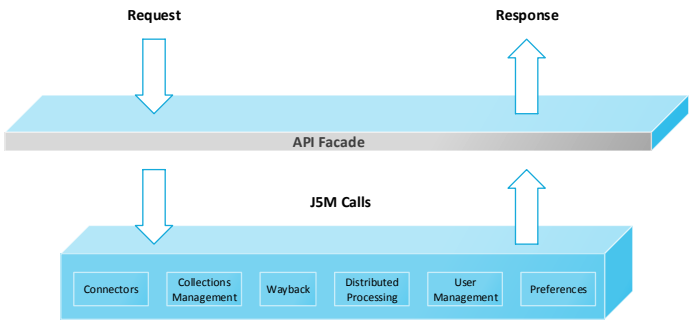
Poslednou oblasťou návrhu systému je správa užívateľských rolí a riadenie prístupu. Motiváciou tejto funkcionality je fakt, že užívatelia vyhľadávacieho systému figurujú na rôznych pozíciach, s ktorými súvisia oprávnenia prístupu k určitým častiam, respektíve funkciám systému. Správa užívateľských účtov bude preto tvorená troma základnými pojmami: užívateľ, skupina a spôsobilosť. Užívateľom chápeme konkrétnu osobu, užívateľský účet, pod ktorým užívateľ v systéme pracuje. Skupina reprezentuje množinu užívateľov. Každá skupina má priradené takzvané spôsobilosti, anglicky capabilities. Spôsobilosť je oprávnenie pristupo-

vať, respektíve používať určite dané systémové funkcie. Konkrétne využitie tohto generického konceptu potom implementuje priamo užívateľské rozhranie, ktoré overuje a sprístupňuje dané funkcie na základe spôsobilostí. Jadro systému bude implementovať výhradne správu užívateľských účtov a priradenie spôsobilostí. Navyše, z pohľadu kolekcií by malo byť možné medzi užívateľmi a skupinami zdieľať prístupu ku kolekciám. Užívatelia by tak získali možnosť vyhľadávať aj v kolekciách iných užívateľov, či celých skupín užívateľov. Implementácia navrhnutého riešenia napĺňa požiadavku 4 z kapitoly 1.2.

Záver: Implementácia jadra webového vyhľadávacieho systému podľa uvedeného návrhu napĺňa požiadavky 2, 3, 5 a 6 z kapitoly 1.2.

3.5 Webové služby

Rozhranie webových služieb navrhovaného vyhľadávacieho systému slúži ako centrálny bod pre komunikáciu nadstavbových aplikácií so službami dostupnými v jadre systému. Primárnym cieľom tejto vrstvy je zjednotenie prístupu k funkciám vyhľadávacieho systému a to štandardizovaným spôsobom tak, aby aplikácie postavené nad týmto rozhraním nemali žiadnu väzbu na vnútornú implementáciu daných funkcionalít. Rozhranie webových služieb bude preto fungovať ako takzvaná API Facade [41]. Využitie tohto návrhového vzoru rieši problém návrhu jednoduchého rozhrania komplexných backednových systémov, kde interné služby a dátové štruktúry nie sú vhodné na priame sprístupnenie pre vývojárov aplikácií externých aplikácií. Obrázok 3.9 ilustruje využitie tejto virtuálnej vrstvy medzi verejným rozhraním a internou implementáciou navrhovaného webového vyhľadávacieho systému.

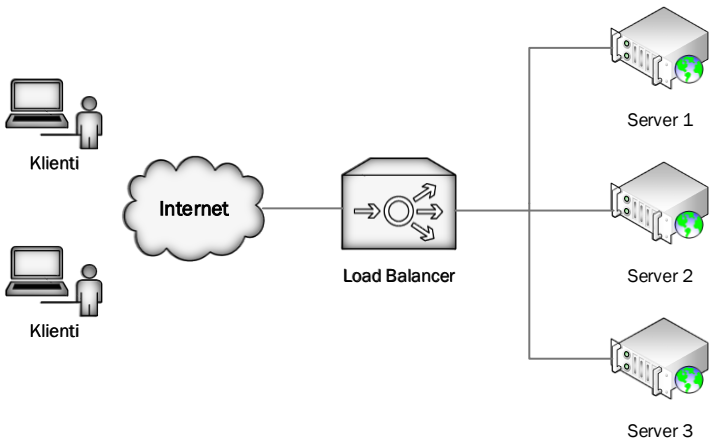


Obr. 3.9: Schéma jednoduchého rozhrania ku komplexnému systému

Aktuálne trendy ponúkajú dva prístupy pre vývoj webových služieb. Konkrétne, tradičnejší prístup využívajúci technológie SOAP [42, 43] a výrazne novší postavený na koncepte zvanom REST [44]. Oba uvedené prístupy majú svoje (ne)výhody, ktoré určujú ich použiteľnosť pre konkrétne nasadenie. Pozrime sa na nich z pohľadu rozhrania webových služieb nášho systému. SOAP je v podstate praotcom všetkých webových služieb. Ide o dobre definované a štandardizované webové služby postavené na zasielaní XML správ v presne definovanom formáte. Jednou z veľkých výhod SOAP-u je generický prenos správ. Narozdiel od REST-u,

ktorý využíva ku komunikácii výhradne protokol HTTP/HTTPS, SOAP nezávisí na konkrétnom transportnom protokole a môže byť tak postavený na HTTP alebo JMS. Naopak, REST je mladým konceptom, ktorý sa neustále rozvíja. Hlavnými prednosťami sú predovšetkým *jednoduchosť* - veľmi rýchla a priamočiara implementácia ako serverovej, tak klientskej časti, *nezávislosť na formáte správ* - narozdiel od SOAPu, ktorý vyžaduje zasielanie komplexných a značne rozsiahlych správ výhradne v XML formáte, REST dovoľuje programátorovi voľbu preferovaného formátu.

Navrhované rozhranie webových služieb by malo byť predovšetkým intuitívne a jednoduché, tak aby umožňovalo rýchly a efektívny vývoj aplikácií. Všetky požiadavky sú dosiahnuteľné implementáciou rozhrania ako RESTful služby. Navyše, súčasné požiadavky na rozhranie nevyžadujú od implementácie žiadnu funkcionalitu, ktorá by nebola dostupná v rámci RESTful služieb a vyžadovala by preto nasadenie komplikovanejšieho riešenia v podobe SOAP-u. Jedinou prekážkou v použití by mohla byť požiadavka na možnosť nasadenia s použitím BPEL [17] technológií, ktoré pracujú primárne s WSDL [28] popisom služieb. Špecifikácia WSDL 1.1 [28] je však určená predovšetkým k popisu SOAP služieb a nie je možné ju použiť k popisu RESTových rozhraní. Uvedený problém ale rieši už verzia WSDL 2.0 [29, 30], ktorá poskytuje aparát na definíciu REST-ových služieb pomocou WEDL deskriptorov, čím umožní nasadenie aj BPEL nástrojov.



Obr. 3.10: Príklad nasadenia load balancera nad rozhraniami webových služieb

Implementácia RESTful webových služieb so sebou prináša ešte jeden významný pozitívny dopad, konkrétne vplyv na škálovateľnosť výsledného riešenia. REST je z definície bezstavový, respektíve stav objektu sa prenáša vrámci URL adresy. Z pohľadu škálovateľnosti riešenia to umožňuje v prípade potreby nasadiť požadovaný počet samostatných inštancií webových služieb a nad týmito inštanciami nastaviť Load Balancing [31], ktorý by napríklad distribuoval požiadavky rovnomerne medzi všetky inštancie, tak ako to ilustruje obrázok 3.10.

Záver: Implementácia rozhrania webových služieb podľa uvedeného návrhu napĺňa požiadavku 7 z kapitoly 1.2.

3.6 Užívateľské rozhranie

Užívateľské rozhranie bude rovnako, ako v projekte WSE [13], slúžiť ako klient nad službami webového vyhľadávacieho systému. Jednou z najväčších nevýhod riešenia WSE je, že sa jedná o monolitickú, len ťažko rozšíriteľnú aplikáciu vo forme webového portálu, ktorá navyše implementuje kompletnú business logiku. Navrhovaný webový vyhľadávací systém bude ale po novom distribuovaný, a výsledné riešenie bude sprístupňovať primárne funkcie prostredníctvom rozhrania webových služieb navrhnutého v kapitole 3.5. Vzhľadom na tieto skutočnosti sa nám ponúka možnosť kompletne prepracovať architektúru užívateľského rozhrania. Nový systém teda počíta s návrhom a implementáciou tenkého klienta vo forme webového portálu, ktorý bude fungovať ako grafické užívateľské rozhranie k webovým službám. Výsledné riešenie teda nebude samo o sebe implementovať žiadnu business logiku. Navyše s ohľadom na budúci rozvoj celého systému, by bolo vhodné implementovať webový portál ako modulárnu, ľahko rozšíriteľnú aplikáciu.

Modularita v rámci portálu by mohla byť dosiahnutá s využitím OSGi [34, 35] technológií.

3.7 Podpora off-line dotazovania

Masívne rozšírenie internetu viedlo k obrovskému nárastu dostupných informácií a potrebe v nich vyhľadávať. Začali tak vznikať prvé fulltextové vyhľadávacie stroje. Trend nárastu množstva informácií sa však neustále zrýchľuje. Štandardné vyhľadávacie algoritmy už nemusia úplne postačovať na vyhľadávanie tých najrelevantnejších dát. Získanie relevantného výsledku na zadaný dotaz môže v určitých prípadoch vyžadovať použitie omnoho sofistikovanejšieho algoritmu výpočtu relevantnosti informácie k zadanému dotazu. V rámci posledného bodu 8 zadania práce, uvedenom v kapitole 1.2, sa preto zamyslíme nad možnosťami, podmienkami a úskaliami návrhu budúcej podpory takzvaného *off-line dotazovania*. Off-line dotazom rozumieme vyhľadávanie relevantných informácií v množine dostupných dát, teda rovnako, ako u bežných dotazov. V tomto prípade je však vyhľadávací algoritmus natoľko výpočetne náročný, že výsledok nie je možné získať v reálnom čase. Naopak, zadaním dotazu sa spustí celý mechanizmus vyhodnocovania, ktorého vykonávanie vyžaduje nezanedbateľné množstvo času a výpočetného výkonu. Typickým príkladom takejto výpočetne a časovo náročnej úlohy je výpočet informačnej vzdialenosti zvolených entít, napríklad tokenov.

V našom každodennom živote nám meranie či odhadovanie priamej vzdialenosti fyzických objektov v podstate nerobí žiaden vážnejší problém. Výpočet vzdialeností objektov vo virtuálnom priestore internetu už však spadá medzi omnoho náročnejšie úlohy. Množinu dokumentov tvoriacich dátovú kolekciu v kontexte nášho vyhľadávacieho systému chápeme ako priestor, v ktorom chceme vypočítavať vzdialenosti medzi objektmi. Konkrétnymi objektmi nášho priestoru sú napríklad slová, vety, odstavce, dokumenty. Uvedené objekty tak v podstate tvoria analógiu reálnemu svetu, kde chceme zistiť vzdialenosti medzi miestami, ulicami, mestami či štátmi. Podobne, ako v reálnom svete, aj v našom priestore

sú jednotlivé objekty spojené cestami a tvoria teda graf. Dĺžka týchto ciest predstavuje informačnú vzdialenosť, ktorú potrebujeme dopočítať. Pozrime sa teraz bližšie na požiadavky takéhoto výpočtu z pohľadu nami navrhovaného systému. Pre podporu off-line dotazov, respektíve výpočet informačnej vzdialenosti musí systém umožniť:

1. export vstupných dát v požadovanom formáte
2. výpočet tranzitívneho uzáveru nad exportovaným grafom

Export vstupných dát pre algoritmus výpočtu tranzitívneho uzáveru je prvým krokom nutným pre podporu off-line dotazov. Graf, ktorý predstavuje kolekciu indexovaných dokumentov a väzby medzi zvolenými entitami, bude samozrejme obrovský a v ďalšej diskusii preto budeme implicitne predpokladať, že sa celý nevojde do pamäte. Najvhodnejšou formou reprezentácie takéhoto grafu bude preto textový súbor. Predovšetkým z dôvodu, že väčšina systémov určených k výpočtu grafových algoritmov vyžaduje na vstupe uvedený formát. Otázkou zostáva ako požadované dáta vygenerovať. Vrámci návrhu systému distribuovaného spracovania dát z kapitoly 3.3 sa ponúka možnosť využiť k implementácii danej úlohy špecifický procesor. Invertizačná rutina projektu Egothor nám totiž implicitne pre každý dokument vygeneruje sekvenciu tokenov, v ňom obsiahnutých, prípadne odkazov na iné dokumenty. Navyše, vďaka ďalším procesorom indexačného procesu máme v rovnakom okamihu prístup k informáciám o referencii a doméne, z ktorej pochádza. V rámci iterácie tokenov vieme tiež okamžite identifikovať každú novú vetu, prípadne nový odstavec. Znamená to, že stačí implementovať relatívne jednoduchý procesor, ktorý prejde celú sekvenciu tokenov každého spracovávaného dokumentu a do požadovaného súboru na disku zapíše väzby medzi entitami v podobe n-tic. Každá takáto n-tica bude obsahovať minimálne term, číslo vety, číslo odstavca. Časová zložitosť takéhoto exportu bude $O(\frac{n}{k})$, kde n je celkový počet spracovávaných dokumentov a k je počet paralelne bežiacich procesorov.

Výpočet tranzitívneho uzáveru je už úloha algoritmicky omnoho náročnejšia ako predošlá príprava dát. Stále rastúci trh spojený so sociálnymi sieťami inšpiroval a inicioval vývoj škálovateľných frameworkov a algoritmov na spracovanie obrovských grafov. Medzi riešenia určené k spracovaniu grafov patria napríklad Pregel [49], GPS [50] prípadne relatívne mladý projekt Giraph [51]. Hlavným problémom dostupných riešení je predovšetkým optimalizácia komunikácie medzi výpočtovými uzlami. Navyše,riešenia, ako napríklad PIG [52], postavené na technike Map&Reduce nie sú vždy optimálne. Konkrétne výpočt tranzitívneho uzáveru vyžaduje niekoľkonásobné opakovanie fázy *map/reduce*. Opakovanie prináša taktiež overhead kvôli nutnosti zlievania výsledkov jednotlivých výpočtových uzlov a nutnosti inicializácie a spustenia nového cyklu. Výpočet tranzitívneho uzáveru s použitím popisovaných riešení by nepatril medzi výkonnostne optimálne, dosiahli by sme ale požadovaného výsledku. Výsledok výpočtu, by vo finále tvoril textový súbor obsahujúci na každom riadku dvojicu termov a váhu príslušnú ceste medzi týmito termami.

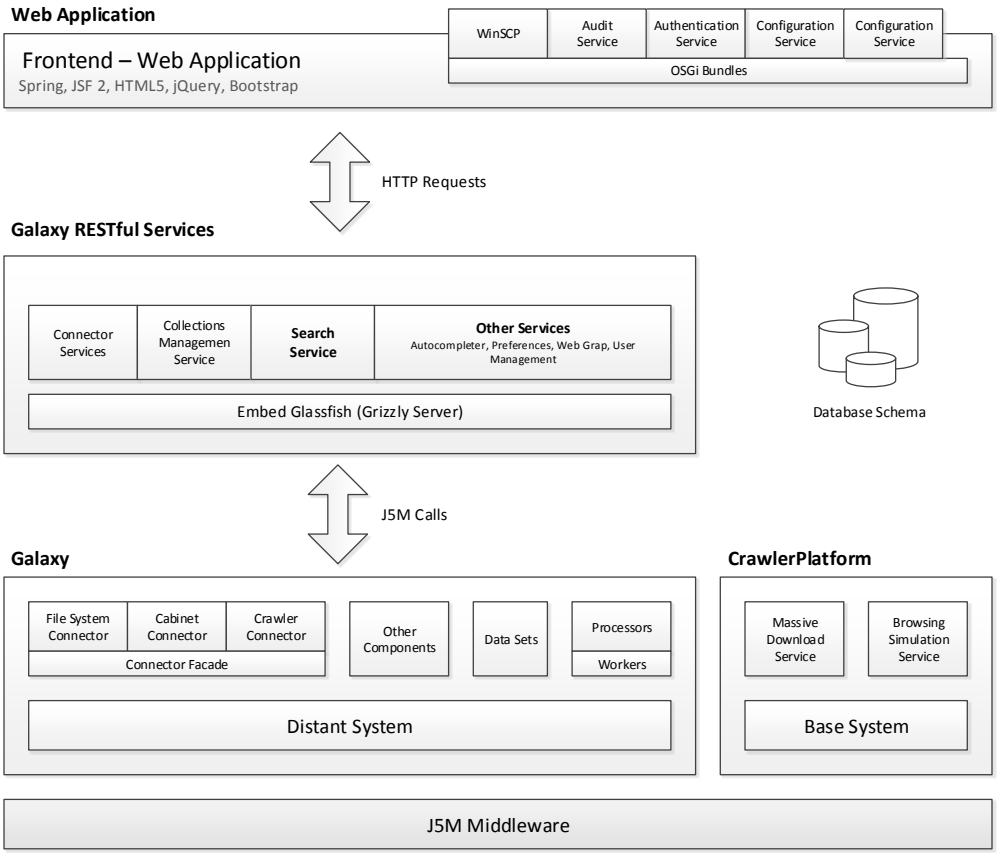
Poznámka: Podpora off-line dotazov predpokladá existenciu, respektíve dostupnosť tranzitívneho uzáveru nad váženým grafom reprezentujúcim kolekciu.

Musíme si ale uvedomiť, že popísaný vstupný graf nie je statický. Mení sa v čase tak, ako sú v rámci systému vkladané, respektíve mazané ďalšie a ďalšie dokumenty. Rámec pre výpočet tranzitívneho uzáveru by mal byť preto schopný pracovať s takzvaným *Time-Evolving Grafom*, v skratke TEG. Systém, ktorý bude schopný pracovať s takýmto typom grafu nebude nútený regenerovať celý tranzitívny uzáver pri každej zmene grafu.

Kapitola 4

Implementácia

Predošlá kapitola popisovala predovšetkým obecný návrh a architektúru požadovaného riešenia tak, aby boli naplnené všetky základné požiadavky na výsledný distribuovaný systém. V nasledujúcich kapitolách postupne predstavíme a popíšeme implementačné špecifiká celého systému. Zameriame sa predovšetkým na vysvetlenia, či odôvodnenia spôsobu konkrétnej implementácie danej funkcionality a z tohto vyplývajúce (ne)výhody.

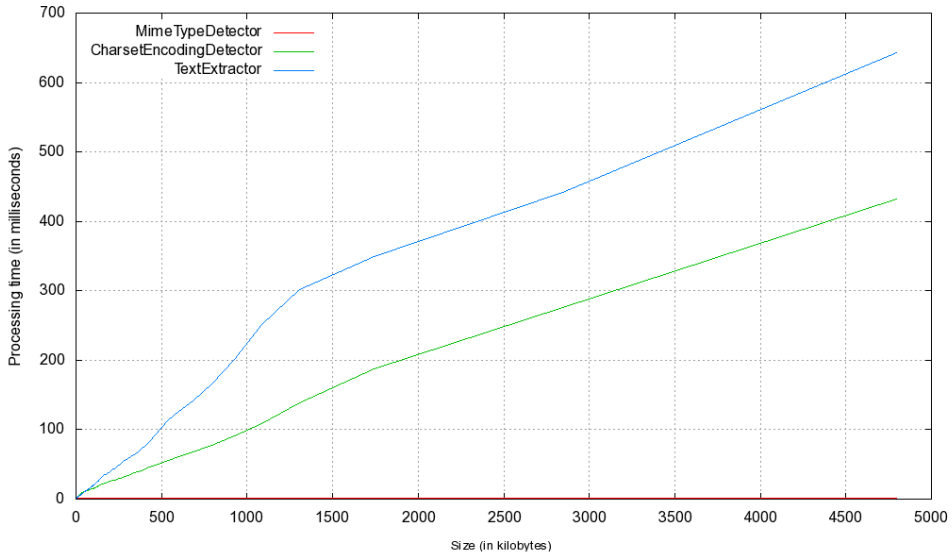


Obr. 4.1: Schéma webového vyhľadávacieho systému

Obrázok 4.1 znázorňuje finálnu implementáciu navrhovaného webového vyhľadávacieho systému.

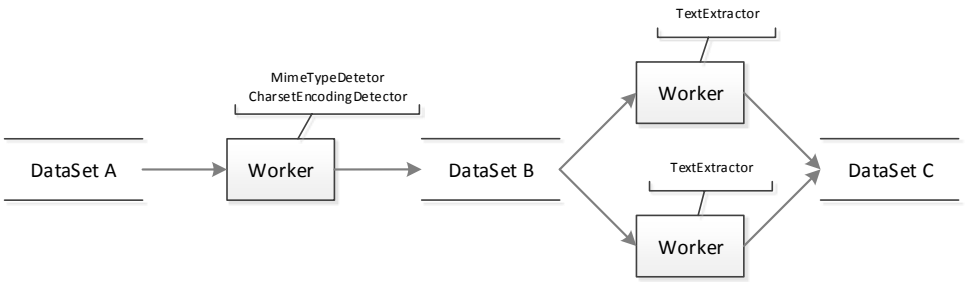
4.1 Distribuované spracovanie dát

Komponenty distribuovaného spracovania dát vykonávajú jedny z výkonnostne najnáročnejších operácií v rámci systému. Medzi konkrétnymi *processormi* sú však značné rozdiely v ich rýchlosti. Obrázok 4.2 porovnáva niekoľko procesorov a ich časové nároky na spracovanie jedného dokumentu, v závislosti na jeho veľkosti. Uvedené tri procesory sme zvolili zámerne, pretože ilustrujú veľkú variabilitu časovej náročnosti konkrétnych rutín.



Obr. 4.2: Porovnanie rýchlosti spracovania medzi procesormi

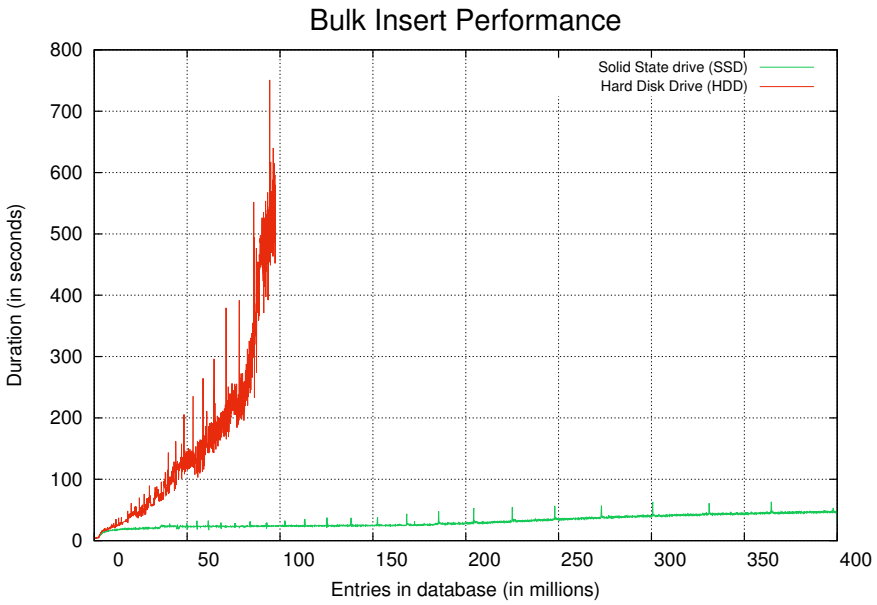
V kapitole 3.3 sme popísali *worker*, ktorý načíta dáta z *datasetu* a spustí nad nimi sekvenciu *processorov*. Celkový počet dokumentov, ktoré je *worker* schopný za jednotku času spracovať, je preto závislí na rýchlosti najpomalšieho *processora* v sekvencii. Riešením je spušťať časovo náročnejšie *processory* paralelne na väčšom počte *workerov*, napríklad ako na obrázku 4.3. Výsledná implementácia totiž umožňuje definovať vlastnú konfiguráciu pre celú skupinu *workerov*. Navyše, ako náhle vznikne požiadavka na urýchlenie spracovania v rámci konkrétnej skupiny, tak stačí pre danú skupinu spustiť novú inštanciu *workera*. Nový *worker* začne okamžite po štarte spracovávať požadované dáta.



Obr. 4.3: Príklad nasadenia *processora* na viacero *workerov*.

4.2 Ukladanie spätných odkazov

Požiadavka 6 z kapitoly 1.2 kládla nárok na možnosť získať k danému dokumentu prehľad o webových stránkach, z ktorých naň vedie odkaz. Výsledné riešenie bolo implementované vo forme databázovej schémy, ktorá obsahuje jednoznačný identifikátor zdroja, cieľa a konkrétnu referenciu vo forme URL adresy zdroja. Skupina testov na veľkých dátových kolekciách zozbieraných robotom za dlhšie časove obdobie ukázala, že priemerný počet odchádzajúcich odkazov z jednej webovej stránky sa pohybuje v rozmedzí 75 - 85 odkazov. Prvá implementácia mechanizmu importu dát do databázy tiež ukázala značnú degradáciu výkonu pri narastajúcom počte záznamov. Systém priebežne vkladal linky do databázy v dávkach po 100 000 záznamov. Obrázok 4.4 znázorňuje značnú degradáciu rýchlosti vloženia jednej dávky v závislosti na celkovom množstve záznamov v databáze. Merania boli vykonané na databázovom serveri PostgreSQL 9.1, ktorého dátové súbory boli umiestnené na štandardnom 10 000 otáčkovom HDD. Príčinou degradácie výkonu pri inserte dávky bola nutnosť aktualizácie indexu nad danou tabuľkou. Pri takomto obrovskom počte záznamov nemôže prebiehať *in-memory* aktualizácia indexu, a tak na degradácii prejavia predovšetkým výkonnostne charakteristiky disku. Vplyv parametrov disk potvrdzuje meranie s použitím SSD disku, ktorý vykazoval rádovo lepšiu výkonnosť. Navyše, operáciu aktualizácie rieši DB server výhradne jediným procesom, ktorý tak nevyťaží všetky CPU.



Obr. 4.4: Degradácia rýchlosti operácie INSERT medzi HDD a SSD diskom.

Hlbšou analýzou možností databázového servera sme dospeli k záveru, že najvhodnejším spôsobom dávkového importu požadovaného množstva dát je použitie SQL príkazu `COPY`. Konkrétne, daný procesor generuje dávkové súbory s počtom 10 000 000 záznamov a tie príkazom `COPY` vloží do databáze. Príkaz je v praxi rádovo niekoľko násobne rýchlejší ako klasický *JDBC bulk insert*.

4.3 Webové služby

Dôležitým faktorom pri implementácii návrhu bol spôsob napojenia webových služieb na podkladovú vrstvu distribuovaného systému. Konkrétne bolo nutné vyriešiť spôsob, akým získajú jednotlivé služby prístup k vzdialeným objektom vyhľadávacieho systému nasadenými v distribuovanom prostredí. Riešením bola implementácia vzdialeného objektu `org.galaxy.services.WebServerImpl`, ktorý sa v rámci distribuovaného prostredia nasadí bežným spôsobom. Uvedený objekt vo fáze inicializácie distribuovaného kontextu inicializuje singleton `org.galaxy.services.RemoteProxy`, na ktorom následne vyvolá metódu `public void setContext(String domain, DistantContext ctx);` a predá mu tak všetky objekty potrebné k implementácii lookup funkcií na prístup k vzdialeným objektom. Inštancia `WebServerImpl` potom po prechode do takzvanej startup fázy, inicializuje a spustí embedovaný Grizzly server. Server pri štarte inicializuje objekty z balíčka `org.galaxy.services.providers.distant`, ktoré sú následne spolu s anotáciou `@org.galaxy.services.annotation.Autowire` využité k dynamickej injekcii referencií na vzdialené objekty priamo do konkrétnej služby.

Štandardným formátom na prenos dát v rámci RESTful služieb implementovaných na platforme Jersey je XML [61]. Najväčšou nevýhodou XML formátu je jeho „ukecanosť“, ktorá pri prenose väčšieho množstva dát značne zvyšuje potrebnú prenosovú kapacitu. Naopak, v súčasnej dobe sa do popredia dostáva JSON [62, 63] formát, ktorý uvedeným problémom netrpí. Spracovanie tohto formátu je veľmi jednoduché, a to predovšetkým pri použití s JavaScriptom. S ohľadom na možný budúci vývoj ďalších aplikácii nad službami vyhľadávacieho jadra bol projekt `galaxy-service` doplnený o možnosť voľby pri výbere dátového formátu komunikácie. Pri štarte webových služieb dôjde k registrácii takzvaného *Content-Negotiation* filtra, ktorý umožní programátorovi vybrať si z dvoch podporovaných formátov, teda XML alebo JSON a to na základe prípony v URL adrese zdroja.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<collection xmlns="http://projects.eblend.net/galaxy">
  <id>1</id>
  <owner>1</owner>
  <name>FileSystem</name>
  ...
  <diskUsage>412707</diskUsage>
</collection>
```

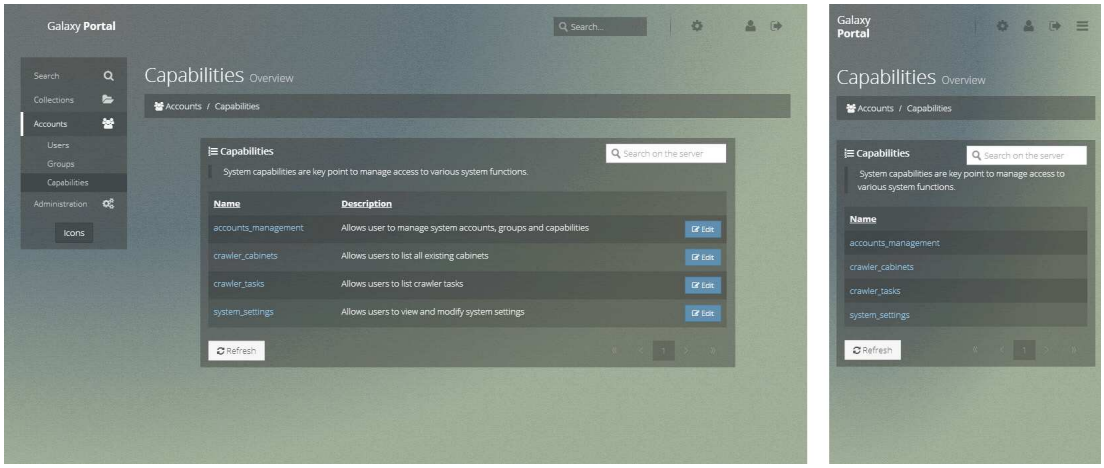
Ukážka 4.1: Príklad odpovede vo formáte XML.

```
{
  "id":"1",
  "owner":"1",
  "name":"FileSystem",
  ...
  "diskUsage":"412707"
}
```

Ukážka 4.2: Príklad odpovede vo formáte JSON.

4.4 Webový portál

Webový portál bol implementovaný ako Java EE webová aplikácia nad rozhraním webových služieb. Súčasťou riešenia je webový portál ako aj sada OSGi balíčkov, ktoré dopĺňajú sadu štandardných funkcií. Prezentačná vrstva bola vytvorená na statickej HTML šablóne s názvom *Light Blue*, ktorá bola zakúpená na stránkach `https://wrapbootstrap.com/`. Licencia k šablóne je súčasťou zdrojových kódov. Základom šablóny je CSS framework Bootstrap. Framework bol vybraný predovšetkým kvôli plánu implementovať užívateľské rozhranie s použitím takzvaného responzivného dizajnu. Motiváciou bol predovšetkým nárast popularity mobilných zariadení a s ním spojená potreba prispôsobovať obsah a formát stránky zariadeniu, na ktorom sa zobrazuje. Obrázok 4.5 znázorňuje plné využitie responsivného designu pri implementácii užívateľského rozhrania.



Obr. 4.5: Zobrazenia stránky pri rozlíšení 1920x1280 a 640x1136 pixelov.

V kapitole 3.2.1 sme navrhli konektor pre vyťažovanie dát z lokálneho súborového systému. Načrtli sme ale taktiež nutnosť umožniť užívateľom import vlastných súborov určených k spracovaniu a indexácii. Požiadavok sme preto vyriešili konceptom takzvaných pracovných adresárov. Každý užívateľ, registrovaný v systéme, má na strane servera k dispozícii vyhradený vlastný pracovný adresár. Adresár slúži predovšetkým k organizácii prístupu k dátam. Akúkoľvek zložku v tomto pracovnom adresári môže definovať ako nový dátový zdroj typu súborový systém, a spustiť tak import jej obsahu do systému. Aby bol užívateľ schopný nahrávať požadované súbory do svojej zložky, prípadne spravovať jej obsah, bol potrebný návrh vhodného mechanizmu. Výsledná implementácia preto obsahuje projekt `galaxy-portal/uloader`. Úlohou balíčka je poskytovať sadu funkcií pre vytváranie a aktualizácie užívateľských účtov na operačnom systéme, konkrétne CentOS, kde je portál nasadený. Vytvorením nového užívateľského účtu na portále sa v rámci operačného systému inicializuje aj nový užívateľský účet spolu s domovským adresárom. Uploader navyše poskytuje funkciu, ktorá vygeneruje ZIP archív s prednastavenou aplikáciou WinSCP pre vzdialený prístup k domovskému adresáru daného účtu. Nahranie vlastných dokumentov na server potom vyžaduje, aby sa užívateľ prihlásil do portálu, stiahol si archív s upladerom, spustil aplikáciu a prihlásil sa s použitím hesla, ktoré sa zhoduje s heslom do webového portálu.

4.5 Zdrojové kódy projektu

Navrhovaný webový vyhľadávací systém, ako aj komponenty prvej generácie pre spracovanie webového obsahu sú implementované v programovacom jazyku Java. Kompletné zdrojové kódy komponent navrhnutých a implementovaných v rámci tejto diplomovej práce sú súčasťou priloženého CD, ktorého obsah je podrobne popísaný v prílohe A. Výsledné riešenie je rozdelené do niekoľkých samostatných projektov a odpovedá logickému rozdeleniu popisovanému v návrhu architektúry, ktorý bol prezentovaný v kapitole 3.

Ako nástroj pre správu, riadenie a automatizáciu buildov bol použitý Apache Maven [53], ktorý je navrhnutý nielen pre uľahčenie práce pri kompilácií aplikácií či ďalších súvisiacich operácií, ale taktiež pre poskytovanie kvalitných informácií o projekte. Základným princípom fungovania Mavenu je popis projektu pomocou Project Object Model [54] definovanom v súbore `pom.xml`, ktorý je vždy umiestnení v koreňovom adresári každého (pod)projektu. Tento model popisuje softvérový projekt nielen z pohľadu zdrojových kódov, ale taktiež z pohľadu závislosti na artefaktoch tretích strán, popisu procesu testovania, kompilácie a publikácie výsledných artefaktov a iných. Navyše, Apache Maven je podporovaný všetkými významnými vývojovými prostrediami pre Javu, ako sú napríklad Eclipse [55], NetBeans [56] či IntelliJ IDEA [57].

4.5.1 Projekt `galaxy`

Projekt `galaxy` predstavuje hlavný agregačný projekt [58] celého výsledného riešenia. Definície uvedené v súbore `pom.xml` predstavujú štandardnú konfiguráciu pluginov, verzií knižníc, licencií, či spôsobu distribúcie softvérových artefaktov a ďalších informácii, ktoré sa implicitne dedia v rámci konfigurácie všetkých modulov. Konkrétne, projekt `galaxy` agreguje následujúcu skupinu modulov:

- `galaxy-processing`, viď kapitola 4.5.2
- `galaxy-core`, viď kapitola 4.5.3
- `galaxy-service`, viď kapitola 4.5.4
- `galaxy-service-beans`, viď kapitola 4.5.5
- `galaxy-rpm`, viď kapitola 4.5.6
- `galaxy-portal`, viď kapitola 4.5.7

Primárnym cieľom a hlavnou výhodou použitia takého agregačného projektu je predovšetkým fakt, že združuje do jednej skupiny všetky implementované moduly a všetky príkazy / operácie vykonáva nad touto skupinou. Navyše, projekt obsahuje taktiež zdrojové kódy pre stránky venované projektovej dokumentácii, do ktorej agreguje aj projektové stránky všetkých definovaných modulov.

Agregovaná projektová dokumentácia, respektíve jej off-line verzia sa generuje spustením príkazu `mvn clean install site:stage` v príkazovom riadku. Výsledná dokumentácia je súčasťou priloženého CD, viď príloha A.

4.5.2 Projekt `galaxy-processing`

Projekt `galaxy-processing` obsahuje implementáciu generickej knižnice určenej k vývoju aplikácii pre distribuované a paralelné spracovanie dát, tak ako bola navrhnutá v kapitole 3.3. Balíček `org.galaxy.processing` predstavený v kapitole 4.5.3 potom názorne ilustruje príklad konkrétneho použitia tohto riešenia. V rámci balíčka `org.galaxy.processing.processors` bola pre potreby indexácie implementovaná rozsiahla skupina procesorov uvedených v prílohe C.1.

4.5.3 Projekt `galaxy-core`

Projekt `galaxy-core` obsahuje implementáciu kompletného jadro webového vyhľadávacieho systému navrhnutého v kapitole 3. Súčasťou projektu je taktiež balíček `org.galaxy.connectors`, ktorý obsahuje implementáciu konektorov podľa návrhu z kapitoly 2. Konkrétne komponenty systému boli implementované v rámci nasledovných balíčkov:

- `org.galaxy.accounts`: Balíček obsahuje systém správy užívateľských účtov a zdieľania prístupu.
- `org.galaxy.autocomplete`: Balíček obsahuje API rozhrania a implementácie služieb takzvaného našepkávača. Jedná sa o službu, ktorá zaznamenáva užívateľmi zadávané dotazy a pre konkrétny dotaz ponúka podobné, najčastejšie hľadané dotazy.
- `org.galaxy.collection`: Balíček obsahuje API rozhrania a implementácie komponent pre management dátových kolekcií a zdrojov.
- `org.galaxy.connectors`: Balíček obsahuje implementáciu konektorov navrhnutých v kapitole 3.2. Konkrétne teda konektory pre súborový systém a vyťažovaciu platformu BOBO.
- `org.galaxy.detection`: Balíček obsahuje API rozhrania a implementáciu tried určených k identifikácii formátu dokumentov a detekcii znakovej sady prípadne jazyka textových dokumentov. Detekciu využíva vyhľadávacie jadro predovšetkým vo fáze spracovania dokumentov.
- `org.galaxy.distant`: Balíček obsahuje implementáciu všetkých vzdialených objektov, ktoré publikujú rozhranie ostatných komponent systému do distribuovaného prostredia.
- `org.galaxy.graph`: Balíček obsahuje implementáciu pre ukladanie spätných odkazov medzi indexovanými dokumentami, predovšetkým však pre webové stránky.
- `org.galaxy.preferences`: Balíček obsahuje API rozhrania a implementácie služby správy užívateľských nastavení.
- `org.galaxy.processing`: Balíček obsahuje konkrétne využitie a nasadenie projektu `galaxy-processing` pre potreby spracovania a indexácie dát v prostredí webového vyhľadávacieho systému. Kompletný zoznam procesorov dostupných v rámci uvedeného balíčka je uvedený v prílohe C.2.

- `org.galaxy.wayback`: Balíček obsahuje API rozhrania a implementácie služieb off-line prístupu k indexovaným dátam a rekonštrukcie webového obsahu.

Poznámka: Všetky implementácie API rozhraní uvedených v predošlom zozname používajú relačnú databázu PostgreSQL [74] ako dátovú vrstvu.

V priebehu implementácie bola taktiež potrebná čiastočná modifikácia niektorých tried z komponent prvej generácie. Zdrojové kódy jadra preto obsahujú aj triedy zaradené mimo hlavný balík `org.galaxy` a to predovšetkým kvôli nutnosti zachovať takzvanú *package visibility* pri aktualizácií kódu. Konkrétne sa to týka balíčkov:

- `kernel.storage`: Obsahuje implementáciu rozhrania `RepositoryReader`, konkrétne novú triedu `TimestampedRepositoryReader`. Trieda popri operáciách, ktoré poskytuje `MultiFileRepositoryReader`, teda extrakcia dát z kabinetu, ukladá do prístupového logu záznamy o prečítaní jednotlivých chunkov daného kabinetu. V praxi to teda znamená optimalizáciu indexačného procesu, pretože pri extrakcii dát `TimestampedRepositoryReader` ignoruje v rámci kabinetu chunky, ktoré boli už predtým prečítané.

- `org.egothor.dir`: Obsahuje dve rozšírenia triedy `TankerImpl`, ktorá vo vyhľadávacom stroji Egothor2 predstavuje štruktúru indexu vrátane základných operácii nad touto štruktúrou, s výnimkou transakcií. Prvou z uvedených je trieda `SharedRepositoryTanker`, ktorej implementácia je takmer úplne totožná s triedou `TankerImplSecure` [22]. Jediným rozdielom je implementácia fáze komitu, kde trieda `SharedRepositoryTanker` operuje so skutočnosťou, že neexistuje rozdiel medzi globálnym a lokálnym repozitárom. Druhým rozšírením je trieda `GalaxyTanker`, ktorá je potomkom triedy `SharedRepositoryTanker` a modifikuje spôsob, akým sa generujú krátke útržky textu dokumentov vo výsledkovej listine. Programátorovi dáva voľnosť vo výbere konkrétnej implementácie `Snipper`.

- `org.egothor.text`: Balíček obsahuje triedy spojené so spracovaním textového obsahu dokumentov. Konkrétne sú dostupné rôzne implementácie triedy `Snipper`, ktorá vytvára krátke útržky textu pre hity výsledkovej listiny. Implementáciou rozhrania `SnipperFactory` a predaním jeho inštancie do konštruktora triedy `GalaxyTanker` potom môže programátor ovplyvňovať / modifikovať algoritmus generovania krátkych útržkov textu. Za zmienku stojí trieda `WildcardSearchSnipper`, ktorá je na rozdiel od iných implementácii rozhrania `Snipper` schopná vrátiť útržky aj pre takzvaný *wildcard search*, viď popis balíka `org.galaxy.egothor`.

4.5.4 Projekt `galaxy-service`

Projekt `galaxy-service` implementuje rozhranie webových služieb nad vyhľadávacím systémom podľa návrhu z kapitoly 3.5. Konkrétne bola zvolená implementácia RESTful služieb postavených s použitím frameworku Jersey [59]. Výsledná aplikácia potom beží vo vnútri JavaEE kontajnera s názvom Grizzly [60].

4.5.5 Projekt `galaxy-service-beans`

Komunikácia medzi užívateľským rozhraním a rozhraním webového portálu projektu `galaxy-portal`, viď 4.5.7 prebieha prostredníctvom protokolu HTTP. Konkrétne, na dátovej úrovni sa predávajú objekty vo formáte XML alebo JSON. Projekt `galaxy-service-beans` obsahuje dátové objekty reprezentujúce vstupné/výstupné parametre všetkých služieb implementovaných v rámci projektu `galaxy-service` a v podstate tak publikuje dátové objekty webových služieb pre použitie v rámci implementácie klienta.

Poznámka: Balík `org.galaxy.services.beans` je plne OSGi-fikovaný a tak pripravený na okamžité nasadenie aj v rámci modulárnych platforiem, akými sú napríklad Felix [64] či Equinox [65], na ktorých je postavený webový portál, viď kapitola 4.5.7.

4.5.6 Projekt `galaxy-rpm`

Nasadenie webového vyhľadávacieho systému na cieľovú platformu vyžaduje nemalé množstvo operácií. Ide nielen o inštaláciu a konfiguráciu nových komponent / knižníc implementovaných počas riešenia tejto diplomovej práce, ale taktiež komponent prvej generácie pre spracovanie webového obsahu, konkrétne J5M middleware, vyťažovaciu platforma BOBO a fulltextový vyhľadávací stroj Egothor2. Projekt `galaxy-rpm` preto pri kompilácií pripraví a vygeneruje inštalačný RPM balíček. Komponenty prvej generácie sú už samostatne distribuované aj vo forme RPM balíčkov, ale s ohľadom na minimalizáciu celkového počtu krokov, ktoré musí užívateľ pri inštalácii vykonať, sme sa rozhodli vytvoriť pre navrhovaný webový vyhľadávací systém jeden inštalačný balíček obsahujúci všetky požadované časti systému:

- komponenty prvej generácie,
- komponenty nového webového vyhľadávacieho systému,
- konfiguračné súbory a skripty,
- a štartovacie skripty

Poznámka: Úspešné vygenerovania RPM balíčka vyžaduje kompiláciu projektu na platforme Linux. Uvedená skutočnosť bola dôvodom, prečo došlo v rámci projektu `galaxy-rpm` k úprave konfigurácie POM tak, aby dochádzalo ku generovaniu RPM iba v prípade kompilácie projektu na platforme Linux.

4.5.7 Projekt `galaxy-portal`

Poslednou komponentou, ktorá vznikla v rámci implementácie vyhľadávacieho systému je projekt `galaxy-portal`. Portál slúži ako ukážkový prototyp aplikácie postavenej nad rozhraním webových služieb backendu a poskytuje základnú sadu funkcií pre prácu užívateľa. Projekt obsahuje sa implementáciu portálu a skupiny samostatných OSGi balíčkou, ktoré poskytujú sadu funkcií používaných v rámci celého systému alebo pridávajú ďalšie vlastnosti.

4.6 Operačné prostredie

Výsledný systém je určený predovšetkým pre nasadenie na Linuxových platformách. Konkrétne, prebiehal celý proces nasadenia a testovania systému na platforme CentOS 6. Z tohto dôvodu boli pre uľahčenie inštalácie a ovládania systému na uvedenom OS vytvorené inštalačné RPM [45, 46, 47] balíčky a dodatočné konfiguračné / štartovacie skripty.

4.6.1 Minimálne hardwarové požiadavky

Pre pohodlnú prácu užívateľa so systémom odporúčame minimálne 2-jadrový procesor 2GHz, aspoň 4 GB RAM a 5 GB HDD. Uvedené minimálne hardvérové nároky sú však veľmi orientačné. Systém bol totiž skúšobne nasadený taktiež v plne virtualizovanom prostredí dátového centra v Prahe. Konkrétne bol systém testovaný na virtuálnom privátnom servery s jedným jadrom s výpočtovou kapacitou 1.5 GHz, 1 GB RAM a 30 GB HDD. Až na očakávanú pomalšiu indexáciu neboli počas testovania zaznamenané žiadne obmedzenia funkčnosti či komfortu práce so systémom.

Vývoj systému prebiehal na stroji 1x Intel Core2Duo 3,0 GHz, 6 GB RAM, 1 TB HDD s operačným systémom Windows 7 64-bit. Testovacie prostredie tvoril výkonný Intel Modular Server [48] s 24 GB RAM, 15 TB HDD s operačným systémom CentOS 6.x 64-bit.

4.6.2 Minimálne softwarové požiadavky

Úspešná inštalácia a následné spustenie implementovaného vyhľadávacieho systému vyžaduje, aby cieľový stroj poskytoval minimálne nasledujúce softvérové vybavenie:

- Operačný systém Linux. Konkrétne bola aplikácia vyvíjaná a testovaná na 64-bitovom operačných systémoch CentOS 6.
- Java Runtime Environment 7 (minimálne update 21).
- Relačná databáza PostgreSQL, verzia 9.1.
- Natívna databáza Berkeley DB, verzia 5.3.21.
- Webový portál vyžaduje nasadenie na aplikačnom serveri podporujúcom technológie OSGi, Jave Servlet a Java Server Faces. Konkrétne bol systém vyvíjaný a testovaný na aplikačnom servery GlassFish v3.2.1 a to pre obe platformy Linux či Windows.
- Webový portál je optimalizovaný na prehliadače Mozilla Firefox (verzia 20 a vyššie) alebo Google Chrome (verzia 28 a vyššie).

Kapitola 5

Návrhy na ďalšie rozšírenie práce

Predošlé kapitoly tejto práce boli venované návrhu a implementácii riešenia, ktoré pokrýva majoritnú časť funkcií a procesov požadovaných od vyhľadávacieho systému. Predstavuje jednoduchý distribuovaný systém schopný spracovania a indexácie zdrojových dát, ako aj ich ďalšiu správu, vyhľadávanie či základnú rekonštrukciu. V neposlednom rade sú všetky funkcie dostupné v integrovanom užívateľskom rozhraní, ktoré tvorí centrálny komunikačný bod medzi užívateľom systému a systémom samotným. S ohľadom na potenciál výslednej implementácie, ako aj jej ďalšieho rozvoja sa v nasledujúcich kapitolách zameriame na priblíženie možností rozšírenia aktuálneho systému. V rámci jednotlivých oblastí nebudeme zachádzať do hlbších detailov prípadného návrhu a implementácie. Pokúsime sa predovšetkým stručne predstaviť problematiku a prípadne zľahka nasmerovať, na niektoré z vhodných riešení danej problematiky.

5.1 Konektory

Základom vyhľadávacieho systému sú dáta, respektíve dátové zdroje, v ktorých systém umožňuje vyhľadávať. Jednoducho povedané, bez dát nie je v čom vyhľadávať. Výsledná implementácia navrhovaného systému obsahuje v predchádzajúcich kapitolách popísaný konektor pre vyťažovanie dát z lokálneho úložiska, ako aj konektor pre webového robota BOBO. Možnosti pre voľbu ďalších zdrojov dát tu ale nekončia. Súčasné OSINTové, respektíve obecné WEBINTové systémy vyžadujú monitoring rozmanitých typov dát. Medzi kľúčové konektory, ktoré by mali byť v budúcnosti doplnené pre potreby tohto projektu patria napríklad emaily, dynamické webové stránky, diskusné fóra, chatové diskusie atď.

5.1.1 Dynamické webové stránky

Dynamické webové stránky, často nazývané aj ako Web 2.0 [66], predstavujú z pohľadu vyťažovania zaujímavú výzvu. Bežné webové stránky sú tvorené samotnou textovou informáciou štruktúrovanou s použitím HTML značiek. Záujmový obsah, text stránky je preto stále dostupný a stačí ho jednoducho extrahovať. Štandardný vyťažovací nástroj sa tak nemusí zaoberať interakciou so stránkou. Stačí stiahnuť HTML kód stránky na danej URL adrese, uložiť obsah, extrahovať odkazy na ďalšej stránky a postup opakovať.

Naopak, moderné dynamické webové stránky sú tvorené predovšetkým obsahom, ktorý je na stránku dynamicky doplňovaný kombináciou AJAX-ových volaní, alebo vyžaduje interakciu s užívateľom. V takomto prípade je samotný HTML kód stránky nositeľom len malého množstva relevantných informácií a obsahuje predovšetkým definície dynamických funkcií, ktoré načítajú obsah až pri ich spustení v prehliadači alebo vyžadujú zásah užívateľa. Existuje hneď niekoľko technológií, ktoré poskytujú funkcionalitu potrebnú k vyťažovaniu dát z takýchto dynamicky generovaných stránok. Základným princípom ich fungovania je automatizovaný prístup k stránkam prostredníctvom internetového prehliadača, tak ako by si stránky prezeral skutočný užívateľ. Koncept simulácie samotného webového prehliadača môže byť dosiahnutý a implementovaný rôznymi spôsobmi. Napríklad projekt HtmlUnit [67, 68] je v podstate internetový prehliadač bez grafického užívateľského rozhrania a má podporu JavaScriptu. Je určený predovšetkým pre integráciu do Java programov, akými sú aj konektory projektu Galaxy. Iným príkladom je projekt WebDriver [69] slúžiaci ako komunikačný prostriedok medzi užívateľským programom a jadrom bežne dostupných prehliadačov, ako Microsoft Internet Explorer, Google Chrome či Mozilla Firefox. Samotný užívateľský program, v tomto prípade vyťažovací nástroj má možnosť simulovať činnosť skutočného užívateľa a zároveň ma prístup k všetkým dátam (elementom na stránke) tak ako by boli prezentované užívateľovi. S použitím uvedených, či iných, technológií je možné implementovať konektor schopný vyťažovať dáta z rôznych sociálnych sietí, diskusných fór. Zaujímavou oblasťou nasadenia by bol napríklad monitoring chatových diskusií.

5.1.2 Emailové účty

Emaily, respektíve emailová komunikácia predstavuje zaujímavý zdroj informácií a to predovšetkým v podnikovej sfére. Z pohľadu zamestnávateľa predstavuje najväčší prínos monitoring emailovej komunikácie a identifikácia potencionálne citlivých informácii, ktoré zamestnanci odosielajú na externé emailové účty. Konektor pre Microsoft Exchange Server [71] by mohol agregovať požadovanú emailovú podnikovú komunikáciu a odosielať ju priamo do vyhľadávacieho systému. Vďaka systému procesorov, ktorý je popísaný v kapitole 3.3.3 by bolo možné vytvoriť a nasadiť novú sadu rutín identifikujúcich citlivé informácie. Kombinácia so službami alertingu z kapitoly 5.3 a extrakcie entit podľa 5.2, by tak užívateľovi umožnila navyše definovať vlastné pravidlá, ktoré by ho upozornili napríklad v prípade, že nejaké citlivé informácie (povedzme čísla kreditných kariet) boli odoslané na externý emailový účet.

Popisovaný konektor monitorujúci komunikáciu priamo na emailovej bráne má síce praktické ale robustné nasadenie. V súvislosti s vyťažovaním emailových schránok sa intuitívne ponúka omnoho kompaktnejšie riešenie. Konkrétne implementácia konektora vyťažujúceho konkrétny emailový účet. Konektor by predstavoval ľahkého klienta, napríklad nad protokolom IMAP [72]. Zadanie monitorovacej úlohy by tak obsahovalo minimálne prístupové údaje a parameter definujúci ako často má konektor kontrolovať a získavať nové správy. Nie menej dôležitým konektorom by bol určite aj konektor pracujúci nad POP3 protokolom.

5.1.3 Java Database Connectivity

Zaujímavou skupinou dátových zdrojov sú relačné databáze. Vzhľadom na fakt, že konektory sú implementované na platforme Java, ponúka sa využitie technológie Java Database Connectivity [73], ďalej len JDBC, k implementácii konektorov na vyťažovanie obsahu z najčastejšie používaných databázových systémov, ako sú napríklad PostrgeSQL [74], MySql [75] či DB2 [76] od spoločnosti IBM. Navrhované konektory by mali pracovať predovšetkým s využitím konfiguračných mechanizmov popísaných v kapitole 3.2. Jedným z možných riešení je využiť túto konfiguráciu k špecifikácií zdrojových tabuliek a pohľadov, spolu s vymedzením polí, ktoré budú obsahovať dáta určené k indexácií. Vyťažovanie by potom mohlo prebiehať tak, že pre každú vyťažovaciu úlohu bude definovaná primárna tabuľka, ktorej každý záznam bude reprezentovať dokument indexovaný v rámci vyhľadávacieho systému. Záznamy zo sekundárnych tabuliek budú môcť byť indexované napríklad ako meta dáta daného dokumentu. Každému primárnemu záznamu bude pred indexáciou priradená unikátna referencia tak, aby existoval jej jednoznačný preklad na konkrétny riadok v tabuľke (vrátane IP adresy servera a mena databáze), z ktorého bola odvodená.

5.1.4 Transkripcia reči

V súčasnosti sa stále viac do popredia dostáva aj transkripcia audio záznamov, ktorá vedie na samotný monitoring rozhlasových či televíznych vysielaní. Medzi popredné firmy v oblasti transkripcie reči či biometriky hlasu patrí napríklad česká firma Phonexia [70]. Zahraničné firmy reprezentuje už v úvode spomínaná firma Autonomy, ktorá v oblasti spracovania audia ponúka produkt s názvom SoftSound Server. Kombinácia transkripcie vysielania a identifikácie hovoriaceho by tak poskytla ďalší bohatý zdroj informácií pre vyhľadávací systém.

5.2 Indentifikácia a extrakcia entít

Z pohľadu webového vyhľadávacieho systému a predovšetkým z pohľadu pridanej hodnoty k spracovávaným informáciám by bola veľkým prínosom identifikácia a extrakcia záujmových entít. Nejedná sa však iba o jednoduché entity typu emailová adresa, či telefónne číslo, ktoré je možné jednoducho identifikovať a extrahovať s pomocou regulárnych výrazov. Zaujímavé sú predovšetkým entity ako osoby, umiestnenia, názvy firiem a to naprieč viacerými jazykmi. Ďalšou zaujímavou nadstavbou by bola možnosť nasadenia nástrojov NLP (Natural Language Processing) [77], ktoré by umožnili identifikovať a extrahovať napríklad väzby medzi nájdenými entitami, sentiment príspevku či iné. Zdrojom inšpirácie by mohli byť projekty OpenNLP [78] a GATE [79].

Výsledné nástroje by následne bolo možne jednoducho začleniť do procesu spracovania dát implementáciou vlastného procesora tak, ako sme ho popísali v kapitole 3.3.3. Procesor by tvoril most medzi systémom distribuovaného spracovania dokumentov a externým NLP systémom. Určené vstupné textové dáta by tak boli odosielané priamo na vstup NLP systému a výstupom by procesor zase obohatil meta dáta spracovávaného dokumentu.

5.3 Monitoring dát a užívateľské upozornenia

Vyhľadávač je dynamicky sa meniacim systémom predovšetkým pri pohľade na dáta. Zvyšujúci sa počet dátových zdrojov, predovšetkým tých on-line, vedie k neustálej indexácii nových dokumentov, prípadne ich novších verzií. V situáciách kedy je primárnou úlohou užívateľa napríklad sledovanie noviniek pre danú tému je nutné manuálne a opakovane vyhľadávať požadovaný dotaz a hľadať nové dokumenty, ktoré do systému pribudli od posledného vyhľadávania. Vhodnejším riešením by bola automatizácia uvedených rutinných činností. Užívateľ by mal k dispozícii takzvaných agentov, u ktorých by definoval požadovaný dotaz a časové intervaly pre vykonávanie dotazov. Aktiváciou agenta by systém v definovaných časových intervaloch automaticky spúšťal zadaný dotaz a hľadal vo výsledkovej listine nové záznamy, ktoré pribudli od poslednej kontroly. V prípade nájdenia nových záznamov by o tejto skutočnosti informoval užívateľa napríklad odoslaním notifikačného emailu.

Kapitola 6

Záver

Táto diplomová práca sa zaoberala návrhom a implementáciou distribuovaného webového vyhľadávacieho systému vychádzajúceho z prvej generácie komponent pre spracovanie webového obsahu. V rámci jednotlivých kapitol sme sa postupne, krok za krokom, venovali predstaveniu problému, základných požiadaviek na výsledný systém, a v neposlednom rade vlastnostiam vstupných komponent, ktoré tvorili odrazový mostík k novému systému. V kapitole 3 boli navrhnuté a popísané všetky kritické komponenty požadovaného riešenia. Navyše, nad rámec zadaných požiadaviek, bol celý systém navrhnutý tak, aby v prípade potreby bolo možné ktorúkoľvek časť jednoducho rozšíriť o nové funkcionality, či nahradiť stávajúce celky novou implementáciou a to bez nutnosti rozsiahlych zásahov do systému.

Výsledkom tejto diplomovej práce je kompletná a plne funkčná implementácia distribuovaného webového vyhľadávacieho systému vrátane modulárneho užívateľského rozhrania. Implementácia plne odpovedá vytvorenému návrhu a napĺňa všetky požiadavky špecifikované u úvode práce, konkrétne v kapitole 1.2. Výsledný systém navyše obsahuje radu funkcií, ktoré neboli súčasťou pôvodného zadania. Dôvodom ich implementácie bola predovšetkým snaha o vytvorenie použiteľného a životaschopného riešenia a nielen naplnenie prvotného zadania. Medzi takéto funkcie patrí predovšetkým rekonštrukcia webového obsahu, vizualizácia zmien v revíziách dokumentov, monitoring činnosti užívateľa, perzistencia užívateľských preferencií či automatické dopĺňanie zadávaných dotazov. Vznikla taktiež samostatná *ready-to-use* nadstavba nad J5M určená predovšetkým k paralelnému a distribuovanému spracovaniu dát tak, ako uvádza kapitola 3.3. V neposlednom rade bol pripravený inštalačný balík, ktorý všetky nutné inštalačné kroky redukuje na vyplnenie niekoľkých konfiguračných parametrov.

S ohľadom na nezanedbateľný potenciál výsledného riešenia sme v kapitole 5 navrhli niekoľko možností, ako na túto prácu nadviazať a obohatiť tak výsledný systém o ďalšie významné funkcie.

Kompletná ukážka živého systému je dostupná priamo s webových stránok projektu `http://projects.eblend.net/galaxy/`.

Zoznam použitej literatúry

[1] Google Search, http://www.google.com

[2] Yahoo Search, http://www.yahoo.com

[3] Bing Search, http://www.bing.com

[4] Google Blog: *We knew the web was big...*, http://googleblog.blogspot.cz/2008/07/we-knew-web-was-big.html

[5] *Enterprise Search*, http://en.wikipedia.org/wiki/Enterprise_search

[6] *OpenSource Intelligence*, http://en.wikipedia.org/wiki/Open-source_intelligence

[7] Autonomy: *IDOL Server*, http://www.autonomy.com/content/Products/products-idol-server/index.en.html

[8] *Tovek Server*, http://www.tovek.cz/produkty-tovek-tovek-server

[9] *Apache Solr*, http://lucene.apache.org/solr/

[10] *Apache Lucene Core*, http://lucene.apache.org/core/

[11] Galamboš, L., *J5M*, 2009.

[12] Galamboš, L., *EGOTHOR 2*, Univerzita Karlova, Praha, http://www.egothor.org/docs/e2.pdf, 2006

[13] Tamáš, M., *Webový vyhľadávač*, 2009.

[14] Galamboš, L., *Bobo: distribuovaný robot*, 2010.

[15] SUHA-COMMONS, https://gforge.cythres.cz/gf/project/suha-commons/.

[16] RFC1738: Uniform Resource Locators (URL), http://www.ietf.org/rfc/rfc1738.txt

[17] Wikipedia: *Business Process Execution Language*, https://en.wikipedia.org/wiki/Business_Process_Execution_Language

[18] *http://www.oracle.com/technetwork/java/javase/tech/index--jsp-136424.html*, http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

[19] *Remote Method Invocation API*, http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/.

[20] Apache River: *JiniTM Architecture Specification*, http://river.apache.org/doc/specs/html/jini-spec.html.

[21] Galamboš, L., J5M: *Installation and Programming Guide*, http://j5m.sourceforge.net/, 2009.

[22] Jakub Podhorný: *Transakce ve fulltextovém vyhledávacím stroji* (Diplomová práca, vedúci: Galamboš L.), MFF UK, Praha, 2007.

[23] Kateřina Dufková: *Dynamická detekce plagiátů* (Diplomová práca, vedúci: Galamboš L.), MFF UK, Praha, 2008.

[24] Martin Pirchala: *Kešovací strategie webových vyhledávacích systémů* (Diplomová práca, vedúci: Galamboš L.), MFF UK, Praha, 2008.

[25] Leo Galamboš: *Egothor 2*, Univerzita Karlova, Praha, http://www.egothor.org/docs/e2.pdf, 2006.

[26] Leo Galamboš, William Sherlock: *Getting Started with ::egothor - Practical Usage and Theory Behind the Java Search Engine*, http://www.egothor.org/book/, 2004

[27] Oracle Berkeley DB, http://www.oracle.com/technology/products/berkeley-db/db/

[28] Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl

[29] Web Services Description Language (WSDL) Version 2.0 Part 0: Primer, http://www.w3.org/TR/wsdl20-primer/

[30] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, http://www.w3.org/TR/wsdl20/

[31] Wikipedia: *Load balancing*, http://en.wikipedia.org/wiki/Load_balancing_(computing)

[32] Oracle Technology Network: *JavaServer Pages Technology*, http://www.oracle.com/technetwork/java/javaee/jsp/index.html

[33] Oracle Technology Network: *JavaServer Faces Technology*, http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html

[34] OSGi Alliance, http://www.osgi.org/

[35] OSGi Alliance: *The OSGi Architecture*, http://www.osgi.org/Technology/WhatIsOSGi

[36] Najork, M., *Querying the Web Graph*. SPIRE, 2010. ISBN 3-642-16320-3.

[37] Ian H. Witten, Alistair Moffat, and Timothy C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Image*. Van Nostrand Reinhold, 1994.

[38] Chakrabarti S., *Mining the Web: Discovering Knowledge from Hypertext Data*. Amsterdam: Morgan Kaufmann, 2003.

[39] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, *Modern Information Retrieval*. Addison Wesley, 1999.

[40] JVM Serializers: *Benchmarking* , `https://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking`

[41] Brian Mulloy: *API Facade Pattern: A Simple Interface to a Complex System*, http://apigee.com/about/content/api-façade-pattern

[42] W3C: *Simple Object Access Protocol*, `http://www.w3.org/TR/soap/`

[43] Wikipedia: *SOAP*, `http://en.wikipedia.org/wiki/SOAP`

[44] Wikipedia: *Representational State Transfer*, `http://en.wikipedia.org/wiki/Representational_state_transfer`

[45] Wikipedia: *RPM Package Manager*, `http://en.wikipedia.org/wiki/RPM_Package_Manager`

[46] RPM.org: *RPM Package Manager*, `http://rpm.org/`

[47] CentOS Deployment Guide: *Using RPM*, `http://www.centos.org/docs/5/html/Deployment_Guide-en-US/s1-rpm-using.html`

[48] Intel: *Intel Modular Server*, `http://www.intel.com/content/www/us/en/modular-server/modular-server.html`

[49] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, *Pregel: a system for large-scale graph processing - abstract* in Proceedings of the 28th ACM symposium on Principles of distributed computing, ser. PODC '09. New York, NY, USA: ACM, 2009, pp. 6–6.

[50] S. Salihoglu and J. Widom, *GPS: A graph processing system*, Stanford University, Technical Report, 2012, `http://ilpubs.stanford.edu:8090/1039/`

[51] Apache Giraph, `http://giraph.apache.org/`

[52] Apache Pig, `http://pig.apache.org/`

[53] Apache Maven, `http://maven.apache.org/`

[54] Apache Maven: *POM Reference*, `http://maven.apache.org/pom.html`

[55] Eclipse: *The Eclipse Foundation open source community website*, `www.eclipse.org`

[56] NetBeans IDE: *The Smarter and Faster Way to Code*, https://netbeans.org/

[57] IntelliJ IDEA: *The Best Java and Polyglot IDE*, http://www.jetbrains.com/idea/

[58] Apache Maven POM Reference: *Aggregation (or Multi-Module)*, http://maven.apache.org/pom.html#Aggregation

[59] Jersey: *RESTful Web Services in Java*, https://jersey.java.net/

[60] Project Grizzly: *NIO Event Development Simplified*, https://grizzly.java.net/

[61] Wikipedia: *NIO Event Development Simplified*, https://grizzly.java.net/

[62] *Introducing JSON*, http://www.json.org/

[63] Wikipedia: *JSON*, http://en.wikipedia.org/wiki/JSON

[64] Apache Felix: *OSGi R4 Service Platform Implementation*, http://felix.apache.org/

[65] Eclipse Equinox: *OSGi R4 Service Platform Implementation*, http://www.eclipse.org/equinox/

[66] Wikipedia: *Web 2.0*, http://en.wikipedia.org/wiki/Web_2.0

[67] HtmlUnit, http://htmlunit.sourceforge.net/

[68] HtmlUnit: *Get Started*, http://htmlunit.sourceforge.net/gettingStarted.html

[69] WebDriver, http://www.w3.org/TR/webdriver/

[70] Phonexia s.r.o: *Speech recognition, voice biometry and HPC products and services*, http://www.phonexia.cz/

[71] Microsoft: *Exchange Server*, http://office.microsoft.com/en-us/exchange/

[72] RFC3501: *Internet Message Access Protocol*, http://tools.ietf.org/html/rfc3501

[73] Oracle Technology Network: *JDBC Overview*, http://www.oracle.com/technetwork/java/overview-141217.html

[74] PostgreSQL: *The world's most advanced open source database*, http://www.postgresql.org/

[75] MySQL: *The world's most popular open source database*, http://www.mysql.com/

[76] IBM: *DB2 database software*, http://www-01.ibm.com/software/data/db2/

[77] Wikipedia: *Natural Language Processing*, https://en.wikipedia.org/wiki/Natural_language_processing

[78] Apache OpenNLP Developer Documentation, http://opennlp.apache.org/documentation/1.5.3/manual/opennlp.html

[79] GATE: General Architecture for Text Engeneering, http://gate.ac.uk/

Zoznam obrázkov

Zoznam použitých skratiek

BOBO	Webový crawler [14]
BPEL	Business Process Execution Language [17]
ES	Enterprise Search
GATE	General Architecture for Text Engineering [79]
IDOL	Intelligent Data Operating Layer [7]
IMAP	Inetrnet Message Access Protocol [72]
J5M	Java 5 Middleware [11]
JINI	Platforma k vývoju distribuovaných aplikácií [20]
JSF	JavaServer Faces [33]
JSON	JavaScript Object Notation [62, 63]
JSP	JavaServer Pages [32]
NLP	Natural Language Processing [77]
Obr.	Obrázok
OSGi	Open Services Gateway initiative [34]
OSINT	Open Source Intelligence [6]
REST	REpresentational State Transfer, [44]
RMI	Remote Method Invocation [18, 19]
SOAP	Simple Object Access Protocol, [42, 43]
SRDB	Systém riadenia báze dát
URL	Uniform Resource Locators [16]
WSDL 1.1	Web Services Description Language, verzia 1.1 [28]
WSDL 2.0	Web Services Description Language, verzia 2.0 [29]
WSE	Web Search Engine [13]

Dodatok A

Obsah CD

Súčasťou diplomovej práce je aj jedno CD, ktoré obsahuje implementáciu riešenia popísaného v tejto práci. Nasledujúci zoznam popisuje obsah jednotlivých adresárov:

- **/dist** – Skompilované knižnice všetkých implementovaných komponent, bez závislostí.
- **/javadoc** – Agregovaná JavaDoc dokumentácia všetkých implementovaných komponent.
- **/measurements** – Namerané dáta a výsledky rôzných testov.
- **/sources** – Zdrojové kódy výsledného riešenia
- **/sources/galaxy-core** – Zdrojové kódy jadra vyhľadávacieho systému
- **/sources/galaxy/galaxy-examples/axis-demo** – Zdrojové kódy ukážky dynamického generovania REST klienta z WSDL popisu.
- **/sources/galaxy-portal** – Zdrojové kódy užívateľského rozhrania, teda webového portálu, vrátane všetkých podprojektov.
- **/sources/galaxy-processing** – Zdrojové kódy nadstavby J5M určenej na implementáciu vlastného paralelného spracovania dát.
- **/sources/galaxy-rpm** – Zdrojové kódy ku kompilácie inštalačného RPM balíčka
- **/sources/galaxy-service** – Zdrojové kódy pre RESTful rozhrania vyhľadávacieho systému
- **/sources/galaxy-services-bean** – Zdrojové kódy dátových objektov pre komunikáciu s RESTful rozhraním
- **/thesis** – elektronická verzia textu tejto práce v PDF formáte

Online verzia projektových stránok je dostupná na domovských stránkach projektu, teda na adrese `http://projects.eblend.net/galaxy/`.

Dodatok B

Komponenty pre spracovanie webového obsahu

Priložený zoznam slúži ako súhrný adresár odkazov na informačné zdroje obsahujúce podrobnejší popis či kompletnú dokumentáciu komponent prvej generácie systému pre spracovanie webového obsahu.

B.1 Java 5 Middleware

- Domovské stránky: `http://www.egothor.org/cms/j5m`
- Projektové stránky: `http://www.egothor.org/product/j5m/`
- Príručka programátora: `http://j5m.sourceforge.net/`

B.2 BOBO - Crawler Platform

- Domovské stránky: `http://www.egothor.org/cms/bobo`
- Projektové stránky: `http://www.egothor.org/product/bobo/`

B.3 Egothor 2

- Domovské stránky: `http://www.egothor.org/cms/egothor2`
- Projektové stránky: `http://www.egothor.org/product/egothor2/`
- Ďalšie informácie: `http://egothor.sourceforge.net/`

B.4 WSE - Web Search Engine

- Domovské stránky: `http://projects.eblend.net/web-search-engine/`

Dodatok C

Textové procesory

Zoznam všetkých aktuálne dostupných procesorov implementovaných pre použitie vo webovom vyhľadávacom systéme. Procesory sú rozdelené do dvoch skupín podľa príslušnosti ku konkrétnemu projektu.

C.1 Projekt `galaxy-processing`

Procesory tu uvedené, tvoria základnú sadu tried pre implementáciu procesorov zameraných na špecifickú úlohu.Všetky uvedené procesory sú lokalizované v balíčku `org.galaxy.computing.processor`

- **`AbstractProcessor`**: Abstraktný procesor od ktorého dedia všetky ďalšie implementácie.
- **`AbstractContextAwareProcessor`**: Abstraktný processor, potomok triedy `AbstractProcessor`, ktorý navyše pridáva sadu štandardných funkcií pre prístup k službám J5M.
- **`BenchmarkedProcessor`**: Abstraktný processor, priamy potomok procesora `AbstractContextAwareProcessor`, ktorý navyše pridáva funkcií pre sledovanie časovej náročnosti procesora.
- **`BenchmarkedProcessorReporter`**: Procesor agreguje merania a generuje prehľad záznamov všetkých potomkov triedy `BenchmarkedProcessor`.
- **`DevNull`**: Zahodí všetky objekty, ktoré má spracovávať.
- **`Identity`**: Nevykonáva žiadnu operáciu.
- **`MultiProcessor`**: Proxy processor, ktorý sa na prvý pohľad chová ako samostatný procesor. Interne ale pozostáva z ľubovoľne dlhej sekvencie iných procesorov.

C.2 Projekt `galaxy-core`

Procesory tu uvedené, tvoria sadu rutín nutných k naplneniu požiadaviek na webový vyhľadávací systém. Všetky uvedené procesory sú lokalizované v balíčku `org.galaxy.processing.processor`.

- **`AbstractFilter`**: Abstraktný procesor, ktorý tvorí základ všetkých procesorov určených k filtrácii dokumentov pri procese importu a indexácie.
- **`AbstractImportFilter`**: Abstraktný procesor, ktorý tvorí základ všetkých procesorov určených k filtrácii dokumentov pri procese importu.
- **`AbstractIndexFilter`**: Abstraktný procesor, ktorý tvorí základ všetkých procesorov určených k filtrácii dokumentov pri procese indexácie.
- **`ChangesDetector`**: Vypočíta percentuálnu zmenu textu dokumentu oproti poslednej známej verzii.
- **`CharsetEncodingDetector`**: Získava informácie o znakovej sade dokumentu.
- **`ChecksumComputer`**: Spočíta kontrolný súčet originálneho dokumentu. Použitím MD5 algoritmus.
- **`CollectionStatisticsUpdater`**: Aktualizuje počet dokumentov aktuálne uložených v kolekcii.
- **`ContentTypeIndexFilter`**: Zabráni indexácii dokumentu na základe jeho formátu.
- **`DeepWebDetector`**: Detekuje vstupný bod do takzvaného deep-webu. Konkrétne detekuje prítomnosť prihlasovacieho formulára na webovej stránke.
- **`FileSizeExtractor`**: Zistí veľkosť originálnych dát.
- **`ImageIndexFilter`**: Zabráni indexácii obrázkov, ktorých rozmery sú menšie ako pod prípustnou hranicou.
- **`ImageMetaDataExtractor`**: Extrahuje meta dáta z obrázkov, typicky EXIF informácie.
- **`Indexer`**: Vloží dokument do fulltextového indexu.
- **`LinksExtractor`**: Extrahuje odkazy na stránke a uloží ich do databáze.
- **`MimeTypeDetector`**: Identifikuje formát dokumentu. Identifikácia môže prebiehať na základe prvých bajtov obsahu dokumentu prípadne na základe prípony. Dostupné sú oba varianty.
- **`OriginalDataDisposer`**: Odstráni pôvodný dokument. Ponechá iba metadáta a samotný textový obsah.
- **`ReferenceCantHaveIndexFilter`**: Zabráni indexácii dokumentov, ktorých referencia obsahuje zadaný reťazec.

- **`ReferenceDomainExtractor`**: Extrahuje z referencie doménu a subdoménu.
- **`TextCantHaveIndexFilter`**: Zabráni indexácii dokumentov ktorých text obsahuje zadaný reťazec.
- **`TextChangesImportFilter`**: Zabráni importu dokumentov, ktorých zmena oproti poslednej dostupnej verzii neprekročila stanovenú minimálnu hranicu.
- **`TextExtractor`**: Extrahuje z dokumentu textovú verziu obsahu. Aktuálne podporuje zhruba 100 rôzných formátov.
- **`TextMustHaveIndexFilter`**: Zabráni indexácii dokumentov ktorých text neobsahuje zadaný reťazec.
- **`TitleSanitizer`**: Doplní chýbajúci názov dokumentu. Získava ho s textu dokumentu.
- **`UUIDGenerator`**: Vygeneruje UUID z referencie.
- **`WaybackBridge`**: Uloží dokument do databáze.
- **`WaybackMetadataSaver`**: Uloží vybrané meta dáta dokumentu do databáze.