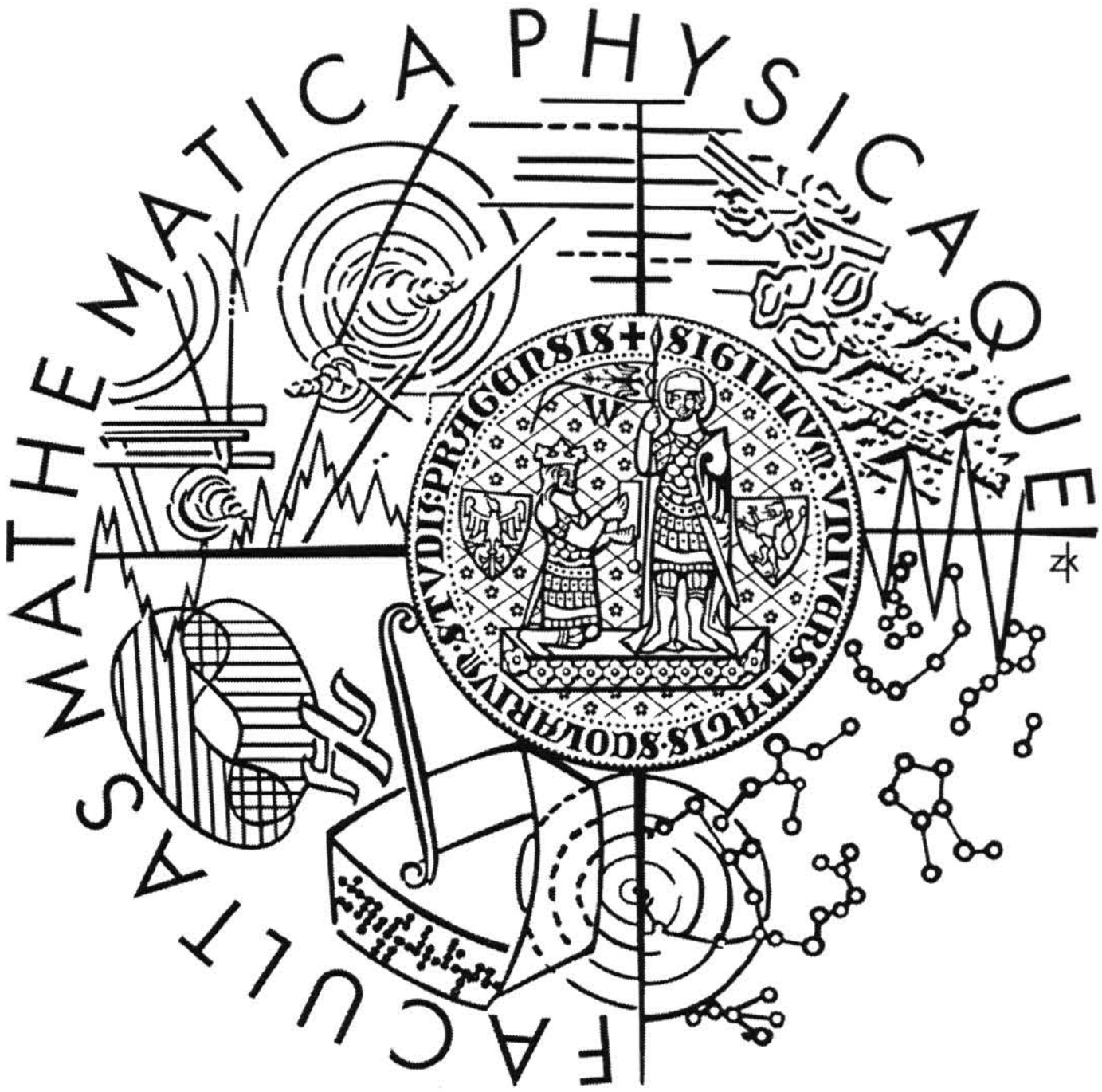
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Michaela Pilátová

Filtrování informací v XML dokumentech

Katedra softwarového inženýrství
Vedoucí diplomové práce: *Prof. RNDr. Jaroslav Pokorný, CSc.*
Studijní program: *Informatika*

Ráda bych poděkovala vedoucímu své diplomové práce Prof. RNDr. Jaroslavu Pokornému, CSc. za usměrňování práce v průběhu jejího vývoje.

Prohlašuji, že jsem svou diplomovou práci napsala samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 13. prosince 2005

Michaela Pilátová

Obsah:

1 Úvod 5

2 Publish-subscribe: od jednoduchých zpráv k filtrování XML 7
2.1 Základní vlastnosti publish-subscribe 7
2.2 Varianty publish-subscribe systémů 8
2.2.1 Topic-based publish-subscribe systémy 8
2.2.2 Content-based publish-subscribe systémy 9

3 XML 11
3.1 XML dokument 12
3.2 Elementy 12
3.3 Stromový model XML dokumentu 13
3.4 Atributy 14
3.5 XML deklarace 14
3.6 Znakové entity 15
3.7 Sekce CDATA 15
3.8 Instrukce ke zpracování 15
3.9 Komentáře 16
3.10 Správně strukturované XML dokumenty 16

4 Práce s XML 17
4.1 XML schémata 17
4.1.1 DTD 17
4.1.2 XML Schema 20
4.2 Parsery 22
4.2.1 Rozhraní DOM 23
4.2.2 Rozhraní SAX 23
4.3 Vyhledávání v XML 24
4.3.1 Cesty 25
4.3.2 Intentifikátory osy 26
4.3.3 Testy uzlu 28
4.3.4 Predikáty 29

5 Publish-subscribe systémy a XML 32
5.1 Volba dotazovacího jazyka 32
5.2 Efektivní vyhodnocování uživatelských dotazů – XML filtrování 33
5.2.1 Architektura filtrovače 34
5.2.2 Od XFilteru k YFilteru 35
5.2.3 Další systémy a přístupy 39
5.3 Publish-subscribe systémy: architektura a konkrétní techniky 40
5.3.1 Základní služby publish-subscribe systémů 40
5.3.2 Routování 42
5.3.3 Přidělování profilů na hostitelské brokery 44
5.3.4 Přenos menší části XML dokumentu 46
5.3.5 Optimalizace přenosu XML dat 51

6 XmlPart - prototypová implementace XML publish-subscribe systému 52
6.1 Nový způsob generování rozpadového schématu 52
6.1.1 Algoritmus 53
6.1.2 Správnost algoritmu 55
6.1.3 Vnořené cesty 57

6.2 Návrh systému 57
6.3 Zjednodušení použitá při implementaci 58
6.4 Instalace systému XmlPart 59
6.5 Ovládání systému XmlPart 59
6.5.1 GUI rozhraní 59
6.5.2 Dávkové programy 64
6.6 Technické řešení 67
6.6.1 Architektura systému 67
6.6.2 Komponenty systému 68
6.6.3 Použité technologie 68
6.7 Experimentální výsledky 68
6.7.1 Porovnání z hlediska času 69
6.7.2 Porovnání z hlediska přenesených dat 70
6.7.3 Porovnání z hlediska přístupu k vnořeným cestám 73
6.7.4 Vyhodnocení experimentů 74

7 Závěr 75

Použitá literatura 76

Dodatek A: Gramatika systému XmlPart 78

Dodatek B: Obsah přiloženého CD-ROM 79

Název práce: *Filtrování informací v XML dokumentech*
Autor: *Michaela Pilátová*
Katedra: *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *Prof. RNDr. Jaroslav Pokorný, CSc.*
e-mail vedoucího: *pokorny@ksi.mff.cuni.cz*
Abstrakt: *S rostoucím množstvím informací dostupných pomocí Internetu a dalších technologií vzrůstá i potřeba distribuovat data jednotlivým uživatelům co nejrychleji, a to jen taková data, o která mají uživatelé zájem. Publish-subscribe systémy umožňují uživatelům pomocí profilů zadaných pomocí vhodného dotazovacího jazyka tento zájem specifikovat a tím objednat příjem požadovaných informací od veškerých zdrojů publikujících do systému.*

K publikování drtivé většiny informací na Internetu se v dnešní době používá jazyk XML. Tato práce se zabývá publish-subscribe systémy pracujícími právě nad jazykem XML. Jsou zde popsány metody efektivní filtrace XML dat a obecné techniky využívané v publish-subscribe systémech k doručení vyhovujících dat jednotlivým uživatelům. Dále je navržen a ve formě prototypu naimplementován vlastní systém XmlPart, jednoduchý publish-subscribe systém pracující nad XML. Pro tento systém je implementováno a na reálných XML datech ověřeno několik konkrétních strategií spojených s doručováním informací.
Klíčová slova: *XML, XPath, filtrování, publish-subscribe systémy*

Title: *Information filtering in XML documents*
Author: *Michaela Pilátová*
Department: *Department of Software Engineering*
Supervisor: *Prof. RNDr. Jaroslav Pokorný, CSc.*
Supervisor's e-mail address: *pokorny@ksi.mff.cuni.cz*
Abstract: *With a growing amount of information available on the Internet and by other technologies, there is a growing need to distribute data to individual users as fast as possible - and only to those who are interested. Publish-subscribe systems allow users to specify their interests using profiles created with an appropriate query language, thus order receiving required information from all available sources contributing to the system.*

The XML language nowadays belongs to the most important standards used for data exchange. This thesis deals with publish-subscribe systems working with data in XML data format. Methods of effective filtration of XML data and general techniques used in publish-subscribe systems for delivery of pertinent data to users are discussed there. Also, a prototype of a simple publish-subscribe system working with XML - XmlPart - is implemented. A couple of techniques connected to information delivery is implemented and tested using this system on real data.
Keywords: *XML, XPath, filtering, publish-subscribe systems*

1 Úvod

Komunikační infrastruktury jako Internet a bezdrátové sítě se v dnešní době stále rozšiřují. Připojuje se čím dál více uživatelů najednou, často pomocí různých komunikačních zařízení. Navíc se spojují různé druhy technologií přístupu k síti (např. mobilní telefony a PDA) a tím vzrůstá potřeba součinnosti mezi různými komunikačními médii. Roste i množství informací a informačních služeb, které jsou uživatelům k dispozici; tento trend se v dohledné době patrně jen tak nezastaví.

V takto velkém množství informací je však již poměrně těžké se orientovat a najít tam informace, které požadujeme, a co je ještě důležitější, najít je rychle. Pokud hledáme nějakou konkrétní informaci, jistě sáhneme po některém z internetových vyhledávačů, těmi se ale v naší práci zabývat nebudeme – budeme se zabývat příbuzným problémem: místo pracného vyhledání informace ve chvíli, kdy ji potřebujeme, budeme chtít, abychom se informaci dozvěděli ve chvíli, kdy vznikne, například proto, abychom na ni mohli zareagovat. To je nutné v oborech, které vyžadují rychlé rozhodování, jako jsou například obchody na burze: těžko budeme každou minutu kontrolovat aktuální kurz našich akcií, když víme, že se bude měnit jen několikrát za den. Místo toho budeme chtít dostat zprávu (např. e-mailem nebo prostřednictvím SMS) ve chvíli, kdy se hodnota akcií změní, proběhne významný obchod apod. Jinými slovy, budeme si chtít objednat příjem určitých (ale asi ne všech) zpráv od jejich poskytovatelů. Systémy, které se tímto procesem zabývají, se zpravidla označují jako *publish-subscribe* (někdy také *vydavatel-předplatitel*) a obecný úvod do této problematiky poskytneme v kapitole 2.

V naší práci se změříme na publish-subscribe systémy, které pracují nad jazykem XML. Tato volba není samozřejmě náhodná – jazyk XML a jeho četné aplikace se v dnešní době používají k publikování drtivé většiny informací na Internetu – je to jednak jazyk XHTML, který se (spolu se svým předchůdcem HTML) používá k tvorbě internetových stránek, dále pak RSS kanály, které používají zpravodajské servery k publikaci přehledu zpráv, a v neposlední řadě také stále populárnější webové služby, které jsou podstatou mnoha B2B (business-to-business) systémů. Další výhodou XML je jednotná syntaxe jazyka, která

umožňuje (díky standardním transformacím) přistupovat jednotným způsobem k datům z různých zdrojů – není tedy podstatné, zda zpráva pochází z CNN nebo ČTK, vždy ji nakonec dostaneme ve stejném tvaru. Jazykem XML a prací s ním se budeme zabývat v kapitolách 3 a 4.

Cílem této práce je poskytnout obecný pohled na problematiku publish-subscribe systémů pracujících zejména se zprávami zapsanými v jazyku XML. Druhým cílem pak je návrh vlastního publish-subscribe systému, jeho implementace v prototypové verzi a ověření na kolekci XML dat.

V kapitole 5 tedy probereme techniky používané v publish-subscribe systémech postavených nad XML. Nejprve se zaměříme na způsoby efektivního výběru požadovaných informací, konkrétně na filtrování XML dokumentů, které díky předzpracování dotazů o daném dokumentu rychle rozhodne, kterým příjemcům je potřeba jej doručit. Další oblastí, kterou prostudujeme, budou techniky související s doručováním požadovaných informací jejich příjemcům.

V kapitole 6 se pak budeme zabývat tvorbou prototypu vlastního publish-subscribe systému XmlPart, pro který navrhneme několik strategií. Popíšeme jednak architekturu systému, jeho ovládání, uvedeme rovněž porovnání jednotlivých implementovaných strategií.

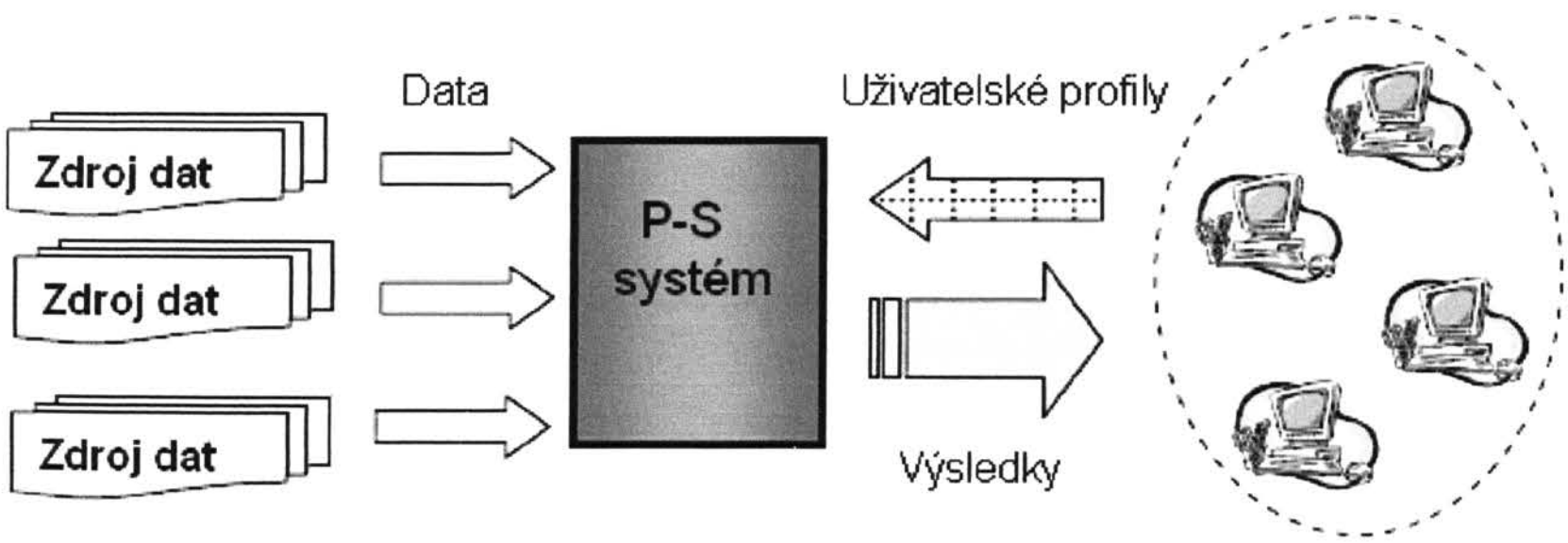
Celou práci poté zakončí závěrečná kapitola 7, ve které krátce shrneme to nejdůležitější z této práce a zhodnotíme její výsledky.

2 Publish-subscribe: od jednoduchých zpráv k filtrování XML

Šíření informací v prostředí velkých sítí může být efektivní pouze tehdy, pokud distribuované informace budou pro uživatele zajímavé. Systémy vyvinuté k efektivnímu šíření informací si musí poradit třeba s miliony uživatelů a zvládnout různorodost jejich požadavků; musí být také schopny filtrovat velké množství informací za jednotku času. Tohoto můžeme dosáhnout pomocí technik využívajících uživatelské profily - dotazy, pomocí kterých uživatel určí, o jaké informace má zájem.

2.1 Základní vlastnosti publish-subscribe

Tyto předpoklady se staly základem pro model *publish-subscribe*. V modelu figurují *poskytovatelé informací* (vydavatelé), kteří publikují zprávy do systému, a *příjemci* (objednatelé, předplatitelé, uživatelé), kteří se přihlásí k odběru informací, které je zajímají.



Obr. 1: Publish-subscribe systém

Poskytovatelé vloží svá data do systému, uživatelé specifikují svůj zájem tím, že systému dodají svůj uživatelský profil, což je dotaz na data zapsaný pomocí nějakého dotazovacího jazyka. Ve chvíli, kdy do systému dorazí nová data od některého z poskytovatelů, všechny registrované uživatelské profily jsou vyhodnoceny vzhledem k těmto nově příchozím datům.

Pouze uživatelům, jejichž profilu zpráva vyhovovala, jsou požadovaná data poslána. Celý proces je schematicky zachycen na obrázku 1.

Velkou výhodou přístupu publish-subscribe je to, že poskytovatelé dat nepotřebují být nijak přímo spojeni s uživateli. Veškerá komunikace je zprostředkována neutrálním prostředníkem, *doručovací službou* (event service, delivery service). Tato služba tudíž umožňuje potlačení následujících vazeb mezi poskytovateli a uživateli:

- *Prostor*: Účastníci se vzájemně nemusí znát. Poskytovatelé i příjemci komunikují pouze s doručovací službou. Poskytovatelé obvykle nemají žádný odkaz přímo k příjemcům, nevědí, kolik těchto příjemců je, nemusí se ani starat o to, aby vůbec nějací byli. Ani uživatelé nemívají o poskytovatelích tyto informace.
- *Čas*: Poskytovatel může publikovat informace v době, kdy je příjemce odpojen, a naopak, příjemce může dostat požadované informace ve chvíli, kdy jejich zdroj je právě odpojen.
- *Synchronizace*: Poskytovatelé nejsou ve chvíli, kdy vkládají své informace do systému, nijak zablokováni. Uživatelé mohou asynchronně přijímat požadovanou zprávu i ve chvíli, kdy jsou zrovna zaneprázdněni něčím jiným.

Odstraněním těchto závislostí se systém stává lépe škálovatelným a přizpůsobeným k použitím v distribuovaných systémech, ke kterým se vrátíme v kapitole 5.

2.2 Varianty publish-subscribe systémů

Příjemci obvykle stojí pouze o některé typy informací, nikoli o všechny informace produkované poskytovateli. Existuje několik variant návrhu publish-subscribe systémů, tyto varianty poskytují různou výrazovou sílu pro situace, kdy chce uživatel vyjádřit zájem o určité informace.

2.2.1 Topic-based publish-subscribe systémy

Původní publish-subscribe systémy, tzv. *topic-based* nebo někdy také *subject-based* (systémy založené na tématu či předmětu), byly založeny na podobném principu, na kterém dnes

fungují např. e-mailové konference. Účastníci mohou poskytovat informace a objednávat příjem informací týkající se konkrétního tématu, které je jednoznačně identifikováno pomocí klíčového slova. Systém bývá v praxi realizován tak, že se jednotlivá témata namapují na odlišné komunikační kanály.

Tento jednoduchý model se může dále zdokonalovat. Jedno z nejužitečnějších vylepšení představuje použití hierarchické struktury témat, které umožňuje témata organizovat na základě jejich vzájemných vztahů. Pokud příjemce požádá o informace týkající se určitého tématu, budou mu doručovány také všechny zprávy spadající pod všechna podtémata. Názvy témat obvykle respektují URL notaci. Většina systémů navíc povoluje použití masek (wildcards), což umožňuje objednání i více různých témat, například všechna témata spadající do jedné konkrétní úrovně hierarchie. Příkladem topic-based systému je TIBCO [19].

2.2.2 Content-based publish-subscribe systémy

Přesto, že obohacení topic-based systémů o užití hierarchií a masek výrazně zvýšilo možnosti těchto systémů, má tento model pouze omezenou výrazovou sílu. Ukázalo se, že je často potřeba se ptát i na samotný obsah zpráv. Můžeme mít například téma o nových knížkách, které bychom chtěli klasifikovat podle nakladatelství, žánru, jazyka, autora atd, některého čtenáře totiž mohou zajímat např. pouze horory a jiného zase pouze knihy psané v angličtině. Toto však již nelze postihnout hierarchickou strukturou, topic-based systémy už tudíž pro tyto účely nestačí. Systémům, které to umožňují, se říká *content-based systémy* (např. Gryphon [2], Siena [4], Jedi [8], Elvin [22]), tj. systémy založené na obsahu. Představují výrazné vylepšení topic-based systémů. Zprávy již nejsou klasifikovány pomocí pouze jediného kritéria (jako byl název tématu), ale podle jejich vlastních vnitřních vlastností, tedy podle obsahu zpráv.

Každá zpráva má v sobě kromě ostatního obsahu také sadu dvojic atribut-hodnota, které obvykle nějakým způsobem popisují obsah této zprávy. Uživatelé svými profily objednávají zprávy tím, že pomocí nějakého dotazovacího jazyka specifikují filtry, kterými musí zpráva úspěšně projít, aby byla příjemci doručena.

Content-based systémy se od sebe navzájem liší mj. vyjadřovací silou jazyka, ve kterém uživatel specifikuje oblast svého zájmu pomocí uživatelských profilů. V nejjednodušších systémech uživatel ve svém profilu jednoduše zadá množinu dvojic - dvojic atribut-hodnota, proti této množině se testuje každá příchozí zpráva. Většinou se lze dotazovat nejen na test na konkrétní hodnotu atributu, ale i na rozsah hodnot. Ve složitějších systémech se navíc mohou mezi filtry používat logické spojky (`AND`, `OR`). Sofistikovanější dotazy vyžadují ještě složitější dotazovací jazyky, např. SQL. Další obecné informace lze najít např. v [11].

S příchodem formátu XML (kapitola 3) se content-based systémům otevřely nové možnosti. Doteď jsme při filtrování zpráv sice mohli použít jejich obsah, ale jen poměrně omezeným způsobem – dvojice atribut-hodnota uvnitř zprávy tvořily jakousi obálku, na kterou jsme se mohli dotazovat, ostatní obsah zpráv zůstal uživatelům skryt až do doručení zprávy. Toto se nyní mění a v publish-subscribe systémech podporujících XML se mohou uživatelé ve svých profilech dotazovat nejen pomocí specifikace atributů a hodnot, ale také se mohou přímo dotazovat i na ostatní obsah zpráv, o které mají zájem, což poté pomůže výrazně vylepšit efektivitu doručování zpráv (uživatel dostane opravdu právě ty zprávy, o které stojí, a žádné navíc). Systémům na bázi XML se budeme dále věnovat v kapitole 5.

3 XML

S růstem významu Internetu jako komunikačního média vyvstala potřeba jednotného formátu pro výměnu dat. Hledal se formát, který by vyhovoval datům, které v sobě nesou jak obsah (text, obrázky, ...), tak i určitou informaci o tom, jakou roli tento obsah v příslušném dokumentu plní - například nadpis kapitoly a popis obrázku má v textu rozdílný logický význam. Strukturovanost takovýchto dat je někde na půli cesty mezi relačními databázemi a zcela nestrukturovanými dokumenty.

Pro řešení tohoto problému se v jednotlivých systémech používaly tzv. značkovací jazyky, tj. jazyky, které obsahují speciální značky (*tagy*) specifikující typ informace, kterou v textu vymezují. Jazyků postavených na tomto principu však bylo příliš mnoho a vzhledem k tomu, že snahou bylo docílit komunikace jednotlivých systémů, musel se jazyk sjednotit. V polovině osmdesátých let minulého století byl přiveden na svět komplexní jazyk SGML, který dovoloval uživateli definovat si vlastní sady značek (a tedy svých vlastních značkovacích jazyků) pomocí speciálních definic (tzv. DTD, viz níže v kapitole 2.3). Ukázalo se však, že kompletní implementace tohoto jazyka je příliš náročná a navíc se v praxi velká část jeho možností nevyužije. Velice populární se stal jednoduchý jazyk HTML, jedna z aplikací SGML, který se používá pro tvorbu webových stránek. Pro naše účely však není dostačující, protože v něm je množina značek i jejich sémantika pevně vymezena a uživatel si tudíž nemůže definovat své značky pro vlastní potřebu.

Jako příjemný kompromis mezi jednoduchostí HTML a komplexností SGML byl navržen jazyk XML (eXtensible Markup Language), který se zanedlouho stal de facto standardem pro výměnu strukturovaných dat po Internetu. Jazyk XML se stal standardem konsorcia W3C [25]. Podrobné informace o jazyku XML lze nalézt v [27], existuje také množství tutoriálů a článků (např. [13]).

V této kapitole bude popsána syntaxe a základní vlastnosti jazyka XML. a v kapitole 4 budou postupně osvětleny další pojmy, které se k XML váží - DTD, XML Schema, parsery SAX a DOM, a dotazovací jazyk XPath.

3.1 XML dokument

Uveďme si rovnou příklad dokumentu ve formátu XML, v dalších podkapitolách si postupně probereme jednotlivé konstrukce.

```
<?xml version="1.0" encoding="utf-8"?>
<zamestnanci>
  <zamestnanec id="101">
    <jmeno>Jan</jmeno>
    <prijmeni>Novák</prijmeni>
    <email>jan@novak.cz</email>
    <email>jan.novak@firma.cz</email>
    <plat>25000</plat>
    <narozen>1965-12-24</narozen>
  </zamestnanec>
  <zamestnanec id="102">
    <jmeno>Petra</jmeno>
    <prijmeni>Procházková</prijmeni>
    <email>prochazkovap@firma.cz</email>
    <plat>27500</plat>
    <narozen>1974-13-21</narozen>
  </zamestnanec>
</zamestnanci>
```

3.2 Elementy

Základní stavební jednotkou každého XML dokumentu je element. V textu je (až na speciální případ prázdného elementu, viz níže) vyznačen a ohraničen počáteční a koncovou značkou (*tagem*). Název značky (také označován jako *typ elementu*) je uzavřen do špičatých závorek, koncová značka má navíc před názvem zpětné lomítko. Mezi těmito značkami je obsah elementu:

```
<zamestnanec>Toto je obsah elementu zamestnanec.</zamestnanec>
```

Element může být i bez obsahu, pak ho zapíšeme tak, že za počáteční značku napíšeme ukončovací:

```
<br></br>
```

Alternativně můžeme použít i zkrácený zápis - za jméno se v počáteční značce zapíše znak '/' a koncová značka se nenapíše vůbec:

```
<br/>
```

Obsahem elementu může být kromě textu i jiný element či elementy, tyto do sebe tedy mohou být vnořené, musí však být utvořeny korektně - musí respektovat stromovou strukturu (viz kapitola 3.3) a počáteční a koncové značky se nesmí křížit. Následuje ukázka dobře a špatně utvořeného elementu:

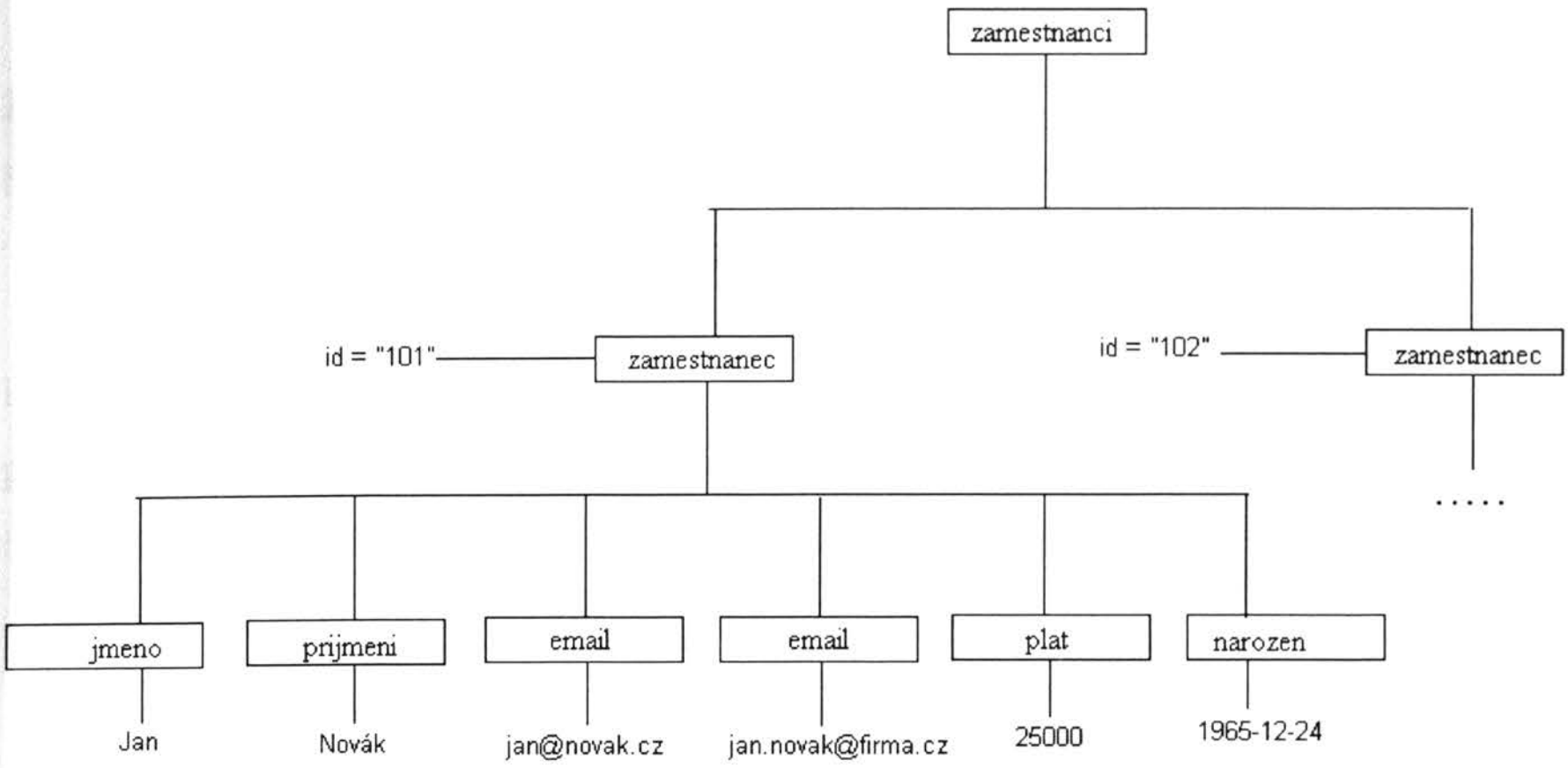
```
<zamestnanec>Toto je obsah elementu zaměstnanec.<br>neco
jineho </br>A tohle taky.</zamestnanec>
```

```
<b>Ukázka <i>překřížení</b> elementů</i>
```

Každý XML dokument musí být navíc celý obsažen v jediném (tzv. kořenovém) elementu.

3.3 Stromový model XML dokumentu

XML dokument si lze představit jako strom, kde vnitřním uzlům odpovídají elementy a listy obsahují vlastní data (textové řetězce), viz obrázek 2 - strom na tomto obrázku odpovídá XML dokumentu z úvodu této kapitoly.



Obr. 2: Stromová reprezentace XML dokumentu

3.4 Atributy

K upřesnění významu elementů a případně k přidání dalších důležitých informací se může (a nemusí) použít jeden či více atributů, které se zapisují do počáteční značky elementu hned za jméno elementu; jsou tvořeny párem atribut-hodnota, přičemž hodnota musí být vždy uzavřena do uvozovek či apostrofů:

```
<zamestnanec id="101">Info o jednom zaměstnanci.</zamestnanec>
```

3.5 XML deklarace

Na začátek dokumentu, ještě před otevírací značku kořenového elementu, můžeme ještě nepovinně připsat tzv. XML deklaraci, která určuje kódování, které dokument používá:

```
<?xml version="1.0" encoding="windows-1250"?>
```

3.6 Znakové entity

Znaky '<' a '>' se používají k oddělení značek a potřebujeme-li je napsat někam do dokumentu bez toho, aby měly tento speciální význam, musíme použít tzv. znakové entity. Zápisu znaku '<' odpovídá entita `<` a pro '>' to je `>`.

Problémy mohou nastat i u jiných znaků: např. pokud potřebujeme pro zápis hodnoty atributu použít uvozovky i apostrofy, využijeme odpovídajících entit `&quot;` a `&apos;`. Jak vidíme, i znak '`&`' má speciální význam; chceme-li tento znak napsat do dokumentu, použijeme entitu `&`. Můžeme si definovat i vlastní entity, viz kapitola 4.1.1.

3.7 Sekce CDATA

Pokud však je v dokumentu úsek, ve kterém se hodně vyskytují znaky se speciálním významem (například v případech, kdy je součástí XML dokumentu kód nějakého programu nebo HTML či XML kód), znakové entity by nebyly příliš pohodlným řešením. Můžeme pro tento případ použít sekce `CDATA`:

```
<script language="JavaScript">
<![CDATA[
  for (i=0; i < 10; $++)
  {
    document.writeln("<p>Ahoj</p>");
  }
]]>
</script>
```

3.8 Instrukce ke zpracování

Občas potřebujeme do textu vložit informace týkající se zpracování textu, přičemž tyto informace nechceme přímo spojovat s obsahem dokumentu. Jedná se například zařazení informací pro různé preprocesory či o vkládání skriptů. Pokud je do dokumentu zařadíme jako *instrukce pro zpracování* (processing instruction), docílíme toho, že je bude XML parser (viz kapitola 4.2) ignorovat a předá je ke zpracování nadřazené aplikaci.

```
...
<dnešníDatum><?php echo Date("d.m.Y")?></dnešníDatum>
...
```

3.9 Komentáře

Je možné také vpisovat komentáře. Tyto jsou součástí dokumentu, ale nejsou nijak zpracovávány. Mohou být v dokumentu kdekoliv s výjimkou vnitřku značek. Zapisují se následujícím způsobem:

```
<!-- Vysvětlující text -->
```

3.10 Správně strukturované XML dokumenty

Pokud dokument splňuje všechna výše uvedená pravidla, je syntakticky v pořádku a říkáme o něm, že je *správně strukturovaný* (well-formed). Pokud je správně strukturovaný, máme zaručeno, že si s ním poradí všechny aplikace podporující formát XML.

4 Práce s XML

V této kapitole budou postupně osvětleny další pojmy, které se k XML váží - XML schémata, parsery XML a dotazovací jazyk XPath.

4.1 XML schémata

Většinou potřebujeme, aby XML dokument kromě správné strukturovanosti splňoval ještě další požadavky. Obrovská síla jazyka XML spočívá v tom, že si můžeme tvořit vlastní jména značek. Avšak bude-li nějaká aplikace nebo parser dokument dále zpracovávat, potřebuje obvykle znát jeho formát poněkud blíž, aby mohly být jednotlivé značky správně interpretovány. Datový formát můžeme definovat pomocí XML schémat, jejich prostřednictvím lze sdělit parseru či aplikacím určité metainformace o dokumentu. Těmito metainformacemi rozumíme např. povolené názvy, pořadí a hnízdění značek, hodnoty atributů a jejich typy či vlastní znakové entity. Pokud je dokument správně strukturovaný a zároveň vyhovuje danému XML schématu, říkáme, že je *validní*. Na dva nejběžnější jazyky používané k popisu XML schématu se nyní podíváme podrobněji.

4.1.1 DTD

Nejdříve uvedeme, jak se v DTD deklarují elementy. Při deklaraci určujeme jméno elementu a model obsahu, čímž specifikujeme, jaké další elementy může náš element obsahovat.

Jméno elementu musí vždy začínat písmenem (nejen z anglické abecedy), následovat mohou písmena, číslice, některé znaky (dvojtečka, pomlčka, podtržítko). Model obsahu specifikuje strukturu elementu. Jako model lze použít klíčová slova `EMPTY` (čímž říkáme, že právě definovaný element už nesmí obsahovat žádné elementy) či `ANY` (tím dáváme najevo, že obsah elementu vůbec neomezujeme). Pro ostatní případy použijeme složitější konstrukci. Do závorek napíšeme všechny podelementy. Pokud mají následovat v přesně stanoveném pořadí, oddělíme je čárkami. Pokud chceme určit, že má následovat pouze jeden z uvedených elementů, oddělíme je pomocí znaku `'|'`. Na stejné úrovni se však nesmí objevit oddělení čárkami a `'|'`, je tedy nutné pečlivě závorkovat:

```
<!ELEMENT abstrakt (nazev, (autor|editor), text_abstraktu)>
```

Za každým z elementů (nebo skupinou elementů v závorkách) v modelové skupině můžeme ještě uvést znak, který určuje počet opakování elementu. Pokud žádný znak uveden není, musí se element objevit právě jednou, '+' znamená aspoň jeden výskyt, '*' znamená libovolné množství výskytů (i nula) a '?' znamená žádný nebo jeden výskyt:

```
<!ELEMENT zamestnanec (jmeno, prijmeni, email+,
                       plat?, adresa*, narozen)>
```

Pokud v elementu nejsou žádné podelementy a jeho obsahem je pouze text, použijeme v modelové skupině text `#PCDATA`:

```
<!ELEMENT jmeno          (#PCDATA)>
```

Pro elementy se smíšeným obsahem (tj. obsahujícími jak text, tak podelementy) lze v modelové skupině kombinovat `#PCDATA` a názvy podelementů:

```
<!ELEMENT odstavec (#PCDATA|tučně|kurzíva)*>
```

Nyní se zaměříme na deklarace seznamu atributů. Deklarace každého z atributů se skládá ze tří částí - jména, typu a výchozí hodnoty. V následujícím příkladě definujeme pro element zamestnanec atribut id:

```
<!ATTLIST zamestnanec
          id      CDATA #REQUIRED>
```

Pravidla pro volbu jména atributu jsou stejná jako ta pro jméno elementu.

Existuje několik možných typů atributu. Častým případem je omezení hodnoty atributu na jednu z předem daných hodnot:

```
<!ATTLIST odstavec zarovnani (vlevo|vpravo|nastred) #IMPLIED>
```

Pokud použijeme jako typ atributu `CDATA`, znamená to, že hodnotou tohoto atributu může být libovolný řetězec. Atribut může také mít hodnotu `ID` - hodnota tohoto atributu pak musí být unikátní v rámci celého dokumentu, tj. `ID` jednoznačně identifikuje příslušný element; každý element může mít pouze jeden atribut tohoto typu. Existuje také komplementární typ `IDREF`

(IDREFS) - hodnotou tohoto atributu musí být ID atributu nějakého elementu v dokumentu (v případě IDREFS je to seznam takovýchto ID atributů oddělený mezerou).

Třetí část deklarace atributu je jeho výchozí hodnota. Napíšeme-li #REQUIRED, znamená to, že atribut vždy musí mít při každém svém výskytu specifikovanou hodnotu. #IMPLIED znamená, že atribut hodnotu mít může a nemusí. Třetí možností je přímo vypsání výchozí hodnoty atributu, tato hodnota se použije v případě, že v dokumentu nebude hodnota tohoto atributu explicitně uvedena.

V DTD lze deklarovat i entity. Ty se požívají, pokud se v dokumentu často opakuje nějaký text. Například, bude-li se v textu často opakovat spojení "Malá mozková příhoda", můžeme si deklarovat entitu &mmp; a tu budeme poté používat namísto dlouhého sousloví:

```
<!ENTITY mmp "Malá mozková příhoda">
```

Uveďme nyní na ukázku DTD, kterému vyhovuje ukázka XML dokumentu z minulé kapitoly:

```
<!ELEMENT zamestnanci (zamestnanec+)>
<!ELEMENT zamestnanec (jmeno, prijmeni, email+,
                       plat?, narozen)>
<!ELEMENT jmeno       (#PCDATA)>
<!ELEMENT prijmeni    (#PCDATA)>
<!ELEMENT email       (#PCDATA)>
<!ELEMENT plat        (#PCDATA)>
<!ELEMENT narozen     (#PCDATA)>
<!ATTLIST zamestnanec
          id     CDATA #REQUIRED>
```

Pokud již máme vytvořený DTD, je třeba ho k dokumentu připojit, aby parser mohl podle něj kontrolovat validnost dokumentu. Obvykle se DTD ukládá do samostatného souboru. Jedno DTD pak můžeme používat v mnoha dokumentech. K dokumentu se DTD připojí pomocí deklarace typu dokumentu (DOCTYPE). Deklarace vždy obsahuje jméno kořenového elementu (v našem případě zamestnanci). Za klíčovým slovem SYSTEM se pak uvádí URL adresa souboru obsahujícího DTD:

```
<!DOCTYPE zamestnanci SYSTEM "zamestnanci.dtd">
<zamestnanci>
  ...
</zamestnanci>
```

Pokud nechceme využít toho, že se jedno DTD může opakovaně používat pro více dokumentů, může být DTD i přímo součástí dokumentu - stačí použít odlišnou deklaraci:

```
<!DOCTYPE zamestnanci [
  <!ELEMENT zamestnanci (zamestnanec+) >
  <!ELEMENT zamestnanec ...>
  ...
]>
<zamestnanci>
  ...
</zamestnanci>
```

Podrobnější informace o schématu DTD lze najít např. v [15].

4.1.2 XML Schema

DTD jako jazyk pro popis schémat oproti svým konkurentům dvě velké výhody. Zaprvé, vzhledem k tomu, že je popsán přímo ve specifikaci XML, ho podporuje obrovské množství aplikací. Druhou velikou výhodou je jeho jednoduchost a přehlednost.

Se stále rychlejším rozvojem XML však začalo být zřejmé, že tento jazyk má také své slabiny:

- Původně měla naprostá většina XML dokumentů textovou povahu - šlo o různé vědecké publikace, články, atd. Později se však formát XML začal používat i pro mnohem strukturovanější data, jako např. kurzové lístky, faktury apod., zde pak už hrají roli i data netextová (měny, peněžní částky, data, ...). K vyjádření datových typů jednotlivých atributů a elementů nejsou v DTD prostředky, existuje v podstatě jediný datový typ - text.
- Další nevýhodou je fakt, že DTD nijak nepodporuje jmenné prostory, tj. v dokumentu XML nemůžeme používat více sad značek..
- Nakonec si ještě uvědomme, že DTD má jiný syntax než XML. Ukázalo se, že je výhodou, když lze se schématem nakládat stejně jako s běžným XML dokumentem. Snad všechny jazyky popisující XML schémata, které přišly až po XML, používají pro zápis syntaxi XML.

DTD je patrně stále nejpoužívanějším jazykem, v poslední době se ale čím dál víc rozmáhá novější jazyk - XML Schema. Je standardizován konsorciem W3C. Tento jazyk odstranil výše zmíněné nevýhody jazyka DTD, i když za cenu toho, že se celý zápis schématu se stal méně přehledným a mnohem složitějším.

Abychom si ukázali syntaxi a hlavní znaky jazyku XML Schema, vraťme se nyní k našemu příkladu XML dokumentu se zaměstnanci:

```
<zamestnanec id="101">
  <jmeno>Jan</jmeno>
  <prijmeni>Novák</prijmeni>
  <plat>25000</plat>
  <narozen>1965-12-24</narozen>
</zamestnanec>
```

Nyní uvedeme příslušné schéma v jazyku XML Schema a následně si ho v krátkosti okomentujeme:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="zamestnanec">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="jmeno" type="xs:string"/>
        <xs:element name="prijmeni" type="xs:string"/>
        <xs:element name="plat" type="xs:decimal"/>
        <xs:element name="narozen" type="xs:date"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Schéma tedy definuje, že dokument má obsahovat element `zamestnanec` s atributem `id` a podelementy konkrétních datových typů.

Všimněme si, že schéma splňuje požadavky na syntaxi XML dokumentů. Jsou v něm použity speciální elementy, které musí patřit do jmenného prostoru, který se označuje obvykle prefixem `xs` nebo `xsd`.

Kořenový element schématu se vždy musí jmenovat `schema` a obsahuje definice elementů (zapisují se pomocí elementu `element`), které se pak mohou v XML dokumentu objevit jako kořenové elementy.

Dále je ve schématu pro každý element specifikován jeho typ. Rozlišují se dva základní druhy typů - jednoduché (řetězec, číslo, datum, ...) a komplexní (elementy komplexního typu mohou obsahovat další elementy nebo atributy). Komplexní typy se označují jako `complexType`, dále pomocí elementu `sequence` specifikujeme sekvenci podelementů. Každý z těchto elementů je povinný, má určené své jméno pomocí atributu `name` a datový typ pomocí atributu `type`. Datové typy se uvádějí jako kvalifikované názvy patřící rovněž do jmenného prostoru XML schémat. Názvy datových typů opět patří do jmenného prostoru schémat. Atributy se deklarují až po elementech, rovněž i v tomto případě musíme specifikovat název a typ atributu.

Pokud nepotřebujeme expresivitu schémat v XML Schema, dáme obvykle přednost zápisu v DTD. Vidíme, že zápis schématu pomocí XMLSchema je oproti DTD mnohem výřečnější a složitější. Možnosti jazyku XML Schema jsou rozsáhlé, další informace viz [15].

4.2 Parsery

Parsery slouží k tomu, abychom mohli rychle zkontrolovat syntaktickou správnost dokumentu. Lze pomocí něj i kontrovat, zda je dokument validní vzhledem k dodanému XML schématu. Programátoři aplikací pracujících s formáty XML se tudíž touto kontrolou nemusí zabývat, pokud využijí některý z parserů v podobě knihoven.

Kromě toho jsou parsery pomocí standardizovaných rozhraní (tzv. API) schopny dokument zpřístupnit v takové podobě, že s ním mohou aplikace jednoduše pracovat. Nejvíce se používají rozhraní SAX a DOM, jsou implementovány ve většině běžně používaných programovacích jazyků – C, C++, Perl, Java, ...

4.2.1 Rozhraní DOM

DOM (Document Object Model) je standardem W3C. V tomto rozhraní je XML dokument modelován jako stromová hierarchická struktura, každému elementu odpovídá jeden uzel stromu. Odpovídající uzly mají samozřejmě i komentáře, veškeré instrukce ke zpracování, apod. Strom tedy uchovává naprosto plnou informaci o dokumentu. Rozhraní DOM obsahuje veškeré potřebné funkce, pomocí kterých lze jednotlivé uzly jakýmkoliv směrem a způsobem procházet a modifikovat, případně uzly mazat či přidávat nové.

Tato reprezentace je velice intuitivní a respektuje přirozenou hierarchickou strukturu XML dokumentů. Je výhodná zejména v aplikacích, ve kterých využijeme možnost přístupu k plné informaci a ve kterých potřebujeme jednotlivé elementy dokumentu procházet a zpracovávat i složitějším způsobem, než pouze ve směru od začátku do konce dokumentu.

Další informace lze dohledat např. v [26] nebo v [14].

4.2.2 Rozhraní SAX

Někdy nám ale naopak onen jednosměrný průchod dokumentem stačí. Pomocí rozhraní SAX (Simple API for XML) můžeme k informacím z XML dokumentu přistoupit nikoli prostřednictvím hierarchického stromu s elementy, jako tomu bylo v případě DOM, ale prostřednictvím sekvence *událostí*. Událostí rozumíme to, že parser při sekvenčním čtení dokumentu např. narazil na otevírací značku nějakého elementu, na obsah elementu, na ukončovací značku elementu, ... Programátor si musí nadefinovat funkce (*handlery*), které budou jednotlivé události obsluhovat, tj. budou nějakým způsobem reagovat na to, že jsme právě vstoupili do elementu.

Rozhraní SAX použijeme např. tehdy, pokud si chceme vytvořit vlastní reprezentaci dokumentu a nevyhovuje nám ta, kterou by nám poskytl DOM, např. proto, že nepotřebujeme uchovávat plnou informaci o dokumentu. Největší výhodou použití SAXu je jeho rychlost a malá spotřeba paměti, neboť nepotřebujeme mít načtený najednou celý dokument, jako tomu bylo u DOMu. Pokud tedy v aplikaci nevyužijeme funkčnosti DOMu, je lépe dát přednost rozhraní SAX.

Samotné rozhraní SAX není definováno pomocí žádného standardu W3C, ovšem představují dnes de facto standard. Bližší informace lze nalézt v [14] nebo [23].

4.3 Vyhledávání v XML

Abychom mohli v XML dokumentu automatizovat vyhledávání informací, potřebujeme k tomu vhodný dotazovací jazyk. Takových pro XML existuje několik - jmenujme XML-QL, XQL či XPath. Nyní se seznámíme podrobněji s posledním z jmenovaných jazyků.

XPath se využívá při práci s XML velmi často, vlastně všude, kde je třeba vyhledávat ve struktuře XML dokumentu určitá data. Chceme-li vybrat množinu dat odpovídajících zadaným podmínkám, obvykle nám postačuje jediný XPath výraz a XML parser se o prohledání dokumentu a vyhodnocení podmínek postará sám.

Zjednodušeně řečeno, XPath především umožňuje vyjádřit relativní cestu od nějakého XML uzlu k jinému elementu nebo atributu. Umí toho mnohem více, ale toto je jeho nejdůležitější funkce. Může tedy připomínat například adresářové cesty v souborových systémech, ovšem je zde podstatný rozdíl - výsledek XPath výrazu (nebo jeho části) může obsahovat jeden, více nebo žádný XML element nebo atribut. Může dokonce obsahovat i jiné datové typy a je tedy mnohem variabilnější.

XPath chápe XML dokument jako strom, který obsahuje uzly následujících typů:

- *Kořenový uzel*: Je umělým uzlem, který neodpovídá žádnému elementu v XML dokumentu. Je rodičem kořenového elementu XML dokumentu, případně i dalších uzlů (instrukce pro zpracování, ...).
- *Elementy*: Každý z elementů XML dokumentu odpovídá samostatnému uzlu stromu. Jeho syni mohou být uzly těchto typů: element, instrukce pro zpracování, textový uzel či komentář. Mohou k němu být navíc připojeny uzly typu atribut či jmenný prostor (pozor, toto připojení není chápáno jako vztah rodič-syn). Název uzlu odpovídá názvu elementu a jeho hodnota odpovídá spojení všech jeho potomků typu textových uzlů.
- *Atributy*: Musí být vždy připojeny ke konkrétním elementům. Ve stromu jsou uzly odpovídající atributům vždy listy. Název uzlu odpovídá názvu atributu a hodnota odpovídá hodnotě atributu.

- *Textové uzly*: Tyto uzly odpovídají textovému obsahu dokumentů. Automaticky jsou v něm již nahrazeny výskyty všech entit příslušným textem (tj., máme-li například ve vstupním dokumentu zapsán odkaz `&lt`, bude v textovém uzlu již zapsán znak '<'). Uzel nemá název a vždy jde o list.
- *Instrukce pro zpracování*: Vždy jde o list, obsahuje cíl instrukce i instrukci jako takovou, názvem je cíl instrukce.
- *Komentáře*: I v tomto případě jde vždy o list. Uzly typu komentář nemají název a jejich obsahem je vždy to, co je uloženo mezi `<!--` a `-->`.
- *Jmenné prostory*: Uzel nese veškeré informace o jmenném prostoru. Vždy jde o list, název odpovídá prefixu daného jmenného prostoru a hodnotou je URI adresa prostoru.

4.3.1 Cesty

Základní součástí jazyka je *path expression*, „výraz popisující cestu". Cesta označuje množinu uzlů v dokumentu. Zapisuje se jako posloupnost přechodů mezi jednotlivými sadami uzlů, ty se oddělují lomítky. Cesty mohou být dvojího typu:

- *absolutní* - strom dokumentu se bude prohledávat od kořene; taková cesta začíná znakem '/'
- *relativní* - dokumenty se prohledávají od aktuálního uzlu

Cesta se zapisuje jako posloupnost kroků mezi jednotlivými sadami uzlů, tyto kroky se oddělují lomítky. Každý krok je určen pomocí tří složek (některé ovšem nemusí být uvedeny, pokud mají implicitní hodnotu):

- *identifikátor osy* – ten určuje, ve kterém směru se budeme od aktuálního uzlu pohybovat
- *test uzlu* – umožňuje vybírat jen některé uzly na základě jejich typu a názvu
- *predikáty* – vybrané uzly můžeme dále filtrovat pomocí podmínek

Příklady:

Nechť je aktuální uzel `<Zamestnanec id="101">` pro dokument reprezentovaný stromem na obrázku 2.

```
jmeno
```

je relativní cesta; vybere uzel `<jmeno> Jan </jmeno>`

```
/zamestnanci/zamestnanec/prijmeni
```

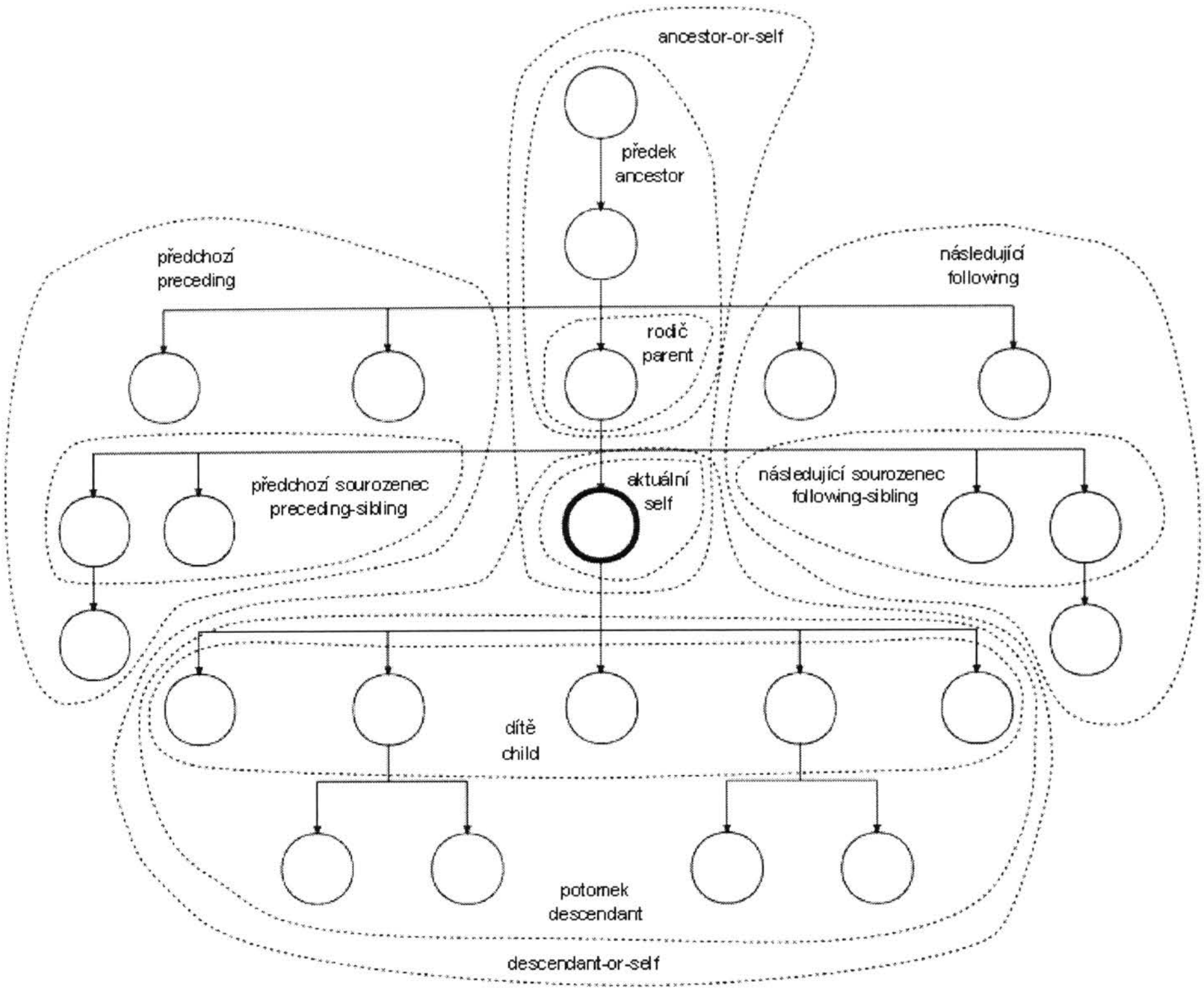
je absolutní cesta; vybere dva uzly - `<prijmeni> Novák </prijmeni>` a `<prijmeni> Procházková </prijmeni>`

4.3.2 Intentifikátory osy

Pohyb po určité ose stromu dokumentu se zapisuje pomocí identifikátoru osy následovaného dvěma dvojtečkami. V následující tabulce vysvětlíme význam jednotlivých druhů os.

Identifikátor	*Vyhodnocené uzly*
`child::`	Přímí potomci aktuálního uzlu.
`descendand::`	Všichni potomci aktuálního uzlu.
`descendand-or-self::`	Aktuální uzel a všichni potomci.
`ancestor::`	Všichni předci aktuálního uzlu.
`ancestor-or-self::`	Aktuální uzel a všichni jeho předci.
`self::`	Aktuální uzel.
`parent::`	Rodič aktuálního uzlu.
`following::`	Všechny uzly, které se v toku XML dokumentu nacházejí za aktuálním uzlem.
`preceding::`	Všechny uzly, které se v toku XML dokumentu nacházejí před aktuálním uzlem.
`preceding-sibling::`	Všichni předcházející sourozenci aktuálního uzlu.
`following-sibling::`	Všichni následující sourozenci aktuálního uzlu.
`attribute::`	Atributy aktuálního uzlu.
`namespace::`	Deklarované jmenné prostory.

Graficky je význam jednotlivých vztahů mezi uzly ve stromové reprezentaci XML dokumentu znázorněn na obrázku 3:



Obr. 3: Vztahy mezi jednotlivými uzly

Příklady:

```
/descendant::zamestnanec/child::jmeno
```

vybere dva uzly - `<jmeno> Jan </jmeno>` a `<jmeno> Petra </jmeno>`

```
/child::zamestnanci/child::zamestnanec/preceding-
sibling::zamestnanec/child::jmeno
```

vybere `<jmeno> Jan </jmeno>`

4.3.3 Testy uzlu

Jak již bylo řečeno, test uzlu dále vymezuje množinu uzlů, která byla určena identifikátorem osy. Testovat můžeme jak název uzlu, tak i jeho typ.

Při hledání elementu či atributu podle názvu zadáme buď název, nebo znak '*', který zastupuje uzel s libovolným názvem:

```
child::para
attribute::lang
child::*
```

Pomocí testu uzlu `prefix:*` můžeme hledat elementy nebo atributy patřící do příslušného jmenného prostoru. Dále máme k dispozici tyto testy:

- `processing-instruction()` – vybere všechny uzly odpovídající instrukci pro zpracování
- `processing-instruction(cíl)` – vybere všechny instrukce pro zpracování s daným jménem
- `comment()` – vybere všechny komentáře
- `text()` – vybere všechny textové uzly
- `node()` – vybere všechny uzly bez ohledu na jejich typ

Vzhledem k tomu, že některé testy uzlů se používají velice často, existuje k nim i zkrácená notace:

- samotný název uzlu je chápán jako pohyb po ose `child::`
- chceme-li se ptát na atribut, lze místo `attribute::` použít zavináč (`@`)
- místo zápisu `self::node()` se lze na aktuální uzel ptát tečkou (`.`)
- podobně také na rodičovský uzel se lze ptát zkráceně dvěmi tečkami (`..`) místo `parent::node()`
- potřebujeme-li prohledávat potomky do libovolné hloubky, lze použít lomítko (`/`) jako zkratku za `descendant-or-self::node()` (čímž budeme mít dvě lomítka za sebou, neboť první z nich se týká oddělování kroků v cestě a druhé specifikaci zmíněných uzlů)

Příklady:

```
//text()
```

vybere všechny textové uzly v celém dokumentu

```
zamestnanec/@id
```

vybere atribut `id` elementu `zamestnanec`, který je dítětem aktuálního uzlu

```
@*
```

vybere všechny atributy aktuálního uzlu

```
zamestnanci//jmeno
```

vybere všechny elementy jmeno, které jsou potomky elementu `zamestnanci`, který je dítětem aktuálního uzlu

4.3.4 Predikáty

V kroku cesty můžeme dále použít predikáty, čímž se znovu zúží množina uzlů, které hledáme. Zapisují se do hranatých závorek a aplikují se na všechny uzly, které až dosud vyhovovaly cestě; dál postupují pouze ty, které vyhovují i predikátu. Predikátů můžeme v jednom kroku použít více, vyhodnocují se zleva doprava.

Výraz použitý v predikátu vrací různé typy:

- číselná hodnota - pokud je výsledkem výrazu v predikátu číslo, chápe se jako pozice v množině uzlů na aktuální ose; pravdivou hodnotu má výraz pouze pro uzly splňující pořadí
- množina uzlů - jako predikát můžeme použít další vnořený kompletní XPath výraz; dále postupují pouze takové uzly, pro které je tento výraz splněn
- další výrazy - jsou převedeny na logickou hodnotu a jejich výsledek je i hodnotou predikátu pro daný uzel.

Ve výrazech predikátů lze používat obvyklé operátory:

- | : sjednocuje výsledky více výrazů

- `+`, `-`, `*`, `div`, `mod` : matematické operátory s obvyklým významem; pro dělení se používá `div` namísto `/`, neboť druhé zmíněné má jiný význam
- `=`, `!=`, `<`, `<=`, `>`, `>=` : relační operátory
- `and`, `or` : logické operátory

XPath rovněž obsahuje veliké množství různých funkcí (je jich přes sto). Uvedeme si ty nejčastěji používané:

Funkce pro práci s uzly:

- `last()` - vrací pozici posledního uzlu mezi kontextovými uzly
- `position()` - vrací pozici aktuálního uzlu mezi kontextovými uzly
- `count()` - vrací počet uzlů v daném seznamu
- `id()` - vrací uzel se zadaným ID
- `name()` - vrací název uzlu

Řetězcové funkce:

- `string()` - převede libovolný objekt na string
- `concat()` - spojí několik řetězců do jednoho
- `contains()` - otestuje, zda řetězec obsahuje hledaný řetězec
- `starts-with()` - otestuje, zda řetězec obsahuje na svém začátku hledaný řetězec
- `substring()` - vrací část řetězce
- `string-length()` - vrací počet znaků řetězce
- `normalize-space()` - odstraní z řetězce přebytečné mezery

Logické funkce:

- `boolean()` - převádí libovolnou hodnotu na logickou hodnotu
- `not()` - vrací negaci výrazu
- `true()` - vrací hodnotu `true`
- `false()` - vrací hodnotu `false`

Funkce pro práci s čísly:

- `number()` - převede libovolný objekt na číselnou hodnotu

- `sum()` - vrací součet hodnot uložených v množině uzlů
- `floor()` - zaokrouhlení na celé číslo dolů
- `ceiling()` - zaokrouhlení na celé číslo nahoru
- `round()` - zaokrouhlení na nejbližší celé číslo

Příklady:

```
/zamestnanci/zamestnanec[1]
```

vybere první element `zamestnanec`, který je dítětem elementu `zamestnanci`, který je kořenovým elementem dokumentu

```
zamestnanec[last()]
```

vybere poslední element `zamestnanec`, který je dítětem aktuálního uzlu

```
/zamestnanci/zamestnanec[jmeno='Jan']
```

vybere element zamestnanec, který je dítětem kořenového elementu `zamestnanci`, pokud obsahuje jako dítě element jmeno s textem `Jan`

```
preceding-sibling::*[1]
```

vybere poslední element, který se nachází před aktuálním uzlem a je na stejné úrovni (je to sourozenec)

```
//cena[@dph='5'][2]
```

vybere druhý element zamestnanec z celého dokumentu, který má atribut `id` nastaven na hodnotu 101

Další informace o jazyku XPath lze najít v [16] či v [28]. Informace o ostatních jazycích lze nalézt např. v [21].

5 Publish-subscribe systémy a XML

Jak jsme již psali v kapitole 1, jazyk XML se rychle začal používat pro poskytování informací prostřednictvím Internetu. Vzhledem k množství a rozmanitosti informačních zdrojů vznikla logická potřeba umět v množství informací vybrat ty, které daného uživatele zajímají. Dostáváme se tak zpět k publish-subscribe systémům, kterými jsme se zabývali v kapitole 2.

Pojďme se nyní podívat, jak tedy vypadá situace na poli publish-subscribe systémů pro dokumenty ve formátu XML. V následujících podkapitolách postupně probereme několik aspektů této problematiky – jaký dotazovací jazyk použít, jak efektivně vyhodnotit množství uživatelských dotazů nad jedním zpracovávaným dokumentem, či jak využít vlastnosti XML dokumentů k omezení množství přenášených dat. Uvedeme také některé nejznámější systémy v této oblasti.

5.1 Volba dotazovacího jazyka

S trochou nadsázky lze říci, že jazyk XML je pro publish-subscribe systémy jako dělaný. A opravdu – díky strukturovanosti dokumentů nemusíme (na rozdíl od content-based systémů založených na atributech) informace, podle kterých je možno se na dokument dotazovat, uvádět ve speciální obálce zprávy, ale přímo ve zprávě samotné. Vyhneme se tím duplikaci dat (atributy, podle kterých vyhledáváme - např. datum, cena, autor, ... - jsou často součástí zprávy) a navíc umožníme příjemcům, aby se ve svých dotazech dotazovali na libovolnou část dokumentu. Další výhodou strukturovaného formátu je, že je poměrně jednoduché (a přirozené) vrátit jen tu část zprávy, která nás zajímá. Tuto funkčnost oceníme např. tehdy, pokud si budeme zprávy posílat na mobilní telefon – zajímat nás asi bude hlavně titulek, kdežto celou zprávu si přečteme, jen pokud nás opravdu bude zajímat. V XML se takové "ořezání" provede snadno – stačí jen vybrat správnou množinu elementů a vrátit podstromy XML dokumentu, které jim odpovídají (zkusme něco takového udělat s nestrukturovaným textem nebo s binárním souborem).

K publish-subscribe systému patří i volba dotazovacího jazyka. Pro XML je přirozenou volbou XPath, kterým jsme se zabývali v kapitole 4.3. Je to jednak díky standardnosti tohoto vyhledávacího jazyka (XPath je obsažen jako základ v téměř v každé XML aplikaci, která provádí dotazování nad XML) a pak také díky množství nástrojů, které s XPath pracují – parsery, procesory, nebo např. debuggery. Přestože jazyk XPath je určen převážně pro vyhledávání, existuje spousta jeho rozšíření, které jej obohacují o nové možnosti – např. XQuery můžeme použít k tomu, abychom nalezená data (vybrané části zprávy) transformovali do podoby, která bude lépe odpovídat našim představám.

Jazyk XPath, tak jak jsme jej představili v kapitole 4.3, má poměrně velkou výrazovou sílu. To je dobré pro komplexní dotazy, které jeho prostřednictvím můžeme pokládat, na druhou stranu je ale pravda, ze složité dotazy se vyhodnocují pomaleji a spotřebují i více paměti. S tím se lze vypořádat v případě dotazování databází dokumentů - tam máme dlouhodobě uložené velké množství dokumentů, které vyhodnocujeme oproti jednomu dotazu. Uvědomme si ovšem, že nyní stojíme před opačným problémem: máme uložené veliké množství dotazů a vyhodnocujeme je v jednu chvíli vždy oproti jedinému dokumentu. Vzhledem k tomu, že publish-subscribe systémy pracují někdy i s miliony dotazů, je množství použitých systémových zdrojů (paměť, procesor) již poměrně podstatným parametrem. Aby se toto množství minimalizovalo, nepoužívají publish-subscribe systémy klasické XPath procesory, ale procesory pracující spíše na principu SAX (klasické procesory potřebují použít DOM, aby mohly vyhodnotit všechny dotazy). Použití procesorů na bázi SAX s sebou nese kromě zrychlení práce i jistá omezení vyjadřovací síly. Zpravidla to znamená omezení se jen na dopředné osy (tj. `child::`, `descendant::` apod.), naopak zpětné osy (`ancestor::`, `preceding-sibling::` apod.) a dotazy do jiné části dokumentu povoleny nejsou. V praxi se ukazuje, že toto omezení zase až tak moc významné není – většina dotazů má již sama onu "dopřednou" povahu a navíc často lze dotazy obsahujících zpětné osy přepsat na dotazy, které obsahují jen osy dopředné - např. dotaz `/a/b/../c` lze přepsat na `/a[b]/c`. V následující podkapitole si ukážeme, jak takové "zjednodušené XPath procesory" fungují.

5.2 Efektivní vyhodnocování uživatelských dotazů – XML filtrování

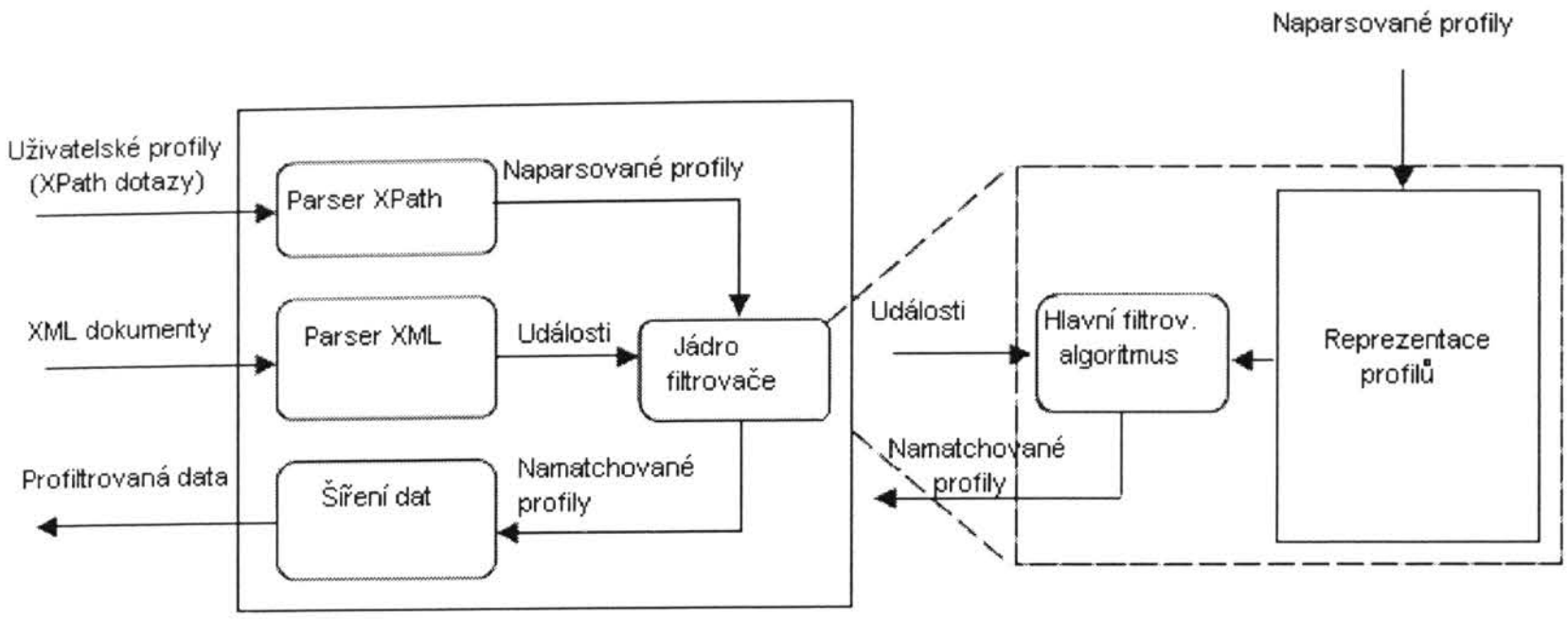
XPath procesory, které rychle zpracovávají zjednodušenou verzi XPath, jak jsme popisovali v předchozí podkapitole, se zpravidla označují jako XML filtrovací systémy, nebo jen XML

filtrovače. To odpovídá jejich funkci – na jeden nebo několik průchodů analyzovaným dokumentem "vyfiltrovat" z dokumentu všechny informace, které nás zajímají. XML filtrovače také zpravidla umějí vyhledat výsledek několika dotazů najednou, což je další z rozdílů proti klasickým XPath procesorům. V této podkapitole si představíme architekturu těchto XML filtrovacích systémů a uvedeme nejpoužívanější přístupy k filtrování XML na příkladech konkrétních aplikací.

5.2.1 Architektura filtrovače

Popíšeme si základní komponenty typického filtrovače (viz obrázek 4).

- *XPath parser*: Komponenta přijme dotaz v XPath, zanalyzuje ho, rozčlení ho na jednotlivé události a pošle do jádra filtrovače.
- *XML parser*: Ve chvíli, kdy do systému přijde nový dokument, XML parser ho zpracuje. Obvykle se používá parser pracující s rozhraním SAX, neboť se tím vyhneme nutnosti načíst celý dokument najednou do paměti (často to ani není možné vzhledem k velikosti dokumentu); dokument je parserem postupně analyzován a rozebírán na jednotlivé události, ve stejné chvíli se dříve zanalyzovaná část může již zpracovávat v jádru filtrovače.
- *Jádro filtrovače*: Když jádro přijme vhodným parserem zanalyzovaný XPath dotaz, přemění ho ve vnitřní reprezentaci. Také naslouchá XML parseru, přijímá od něj jednotlivé události a reaguje na ně pomocí naimplementovaných handlerů, které vykonají vlastní filtraci. Výstupem jsou identifikátory dotazů, kterým XML dokument odpovídal.



Obr. 4: Základní komponenty filtrovače

Obvykle filtrovací systémy podporují jak dotazy na strukturu dokumentu (do určité míry, důvod viz kapitola 5.1), tak i na predikáty. Nicméně snad každý filtrovač se na některou z těchto strategií se zaměřil a vyhledává primárně buď podle struktury nebo podle predikátů. Typický filtrovač je prvního typu, tj. preferuje strukturu, nejznámějším dvěma zástupcům takových filtrovačů se budeme věnovat v následující kapitole 5.2.2, několik dalších filtrovačů krátce představíme v kapitole 5.3.3.

5.2.2 Od XFilteru k YFilteru

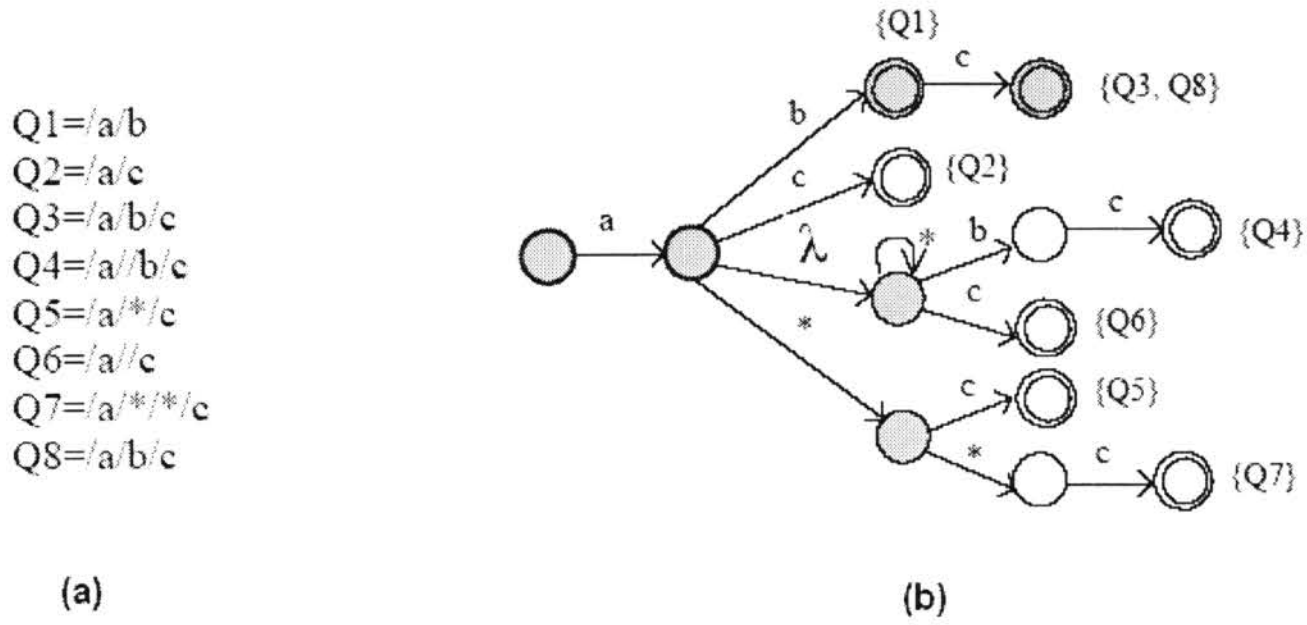
Pro jakýkoli výraz v XPath, který používá pouze osy `child::` a `descendant::` a testy uzlů specifikovaných buď názvem elementu nebo maskou '`*`', lze sestrojit konečný automat, který přijímá jazyk popsaný tímto výrazem (podrobnosti viz [12]). Lze tedy pro každý dotaz vytvořit vlastní konečný automat a hlídat, u kterého z dotazů dosáhneme koncového stavu. Průkopníkem této myšlenky je XFilter [1]. Tento systém se však již zdá být překonaný - XFilter nijak dotazy nijak nezkoumá a nevyužívá jejich podobnosti.

Je-li dotazů hodně, využití příbuznosti dotazů může výrazně vylepšit výkon systému. YFilter [9] vychází ze systému XFilter nastíněného v kapitole 5.2.1, představuje však jeho významné vylepšení. Jedná se o patrně nejznámější a zatím nepřekonaný filtrovací systém, jeho myšlenky si nyní přiblížíme důkladněji.

Podobně jako XFilter, i YFilter využívá možnosti reprezentovat dotazy pomocí konečného automatu. Ovšem zatímco XFilter používal na každý dotaz jeden konečný automat a tyto

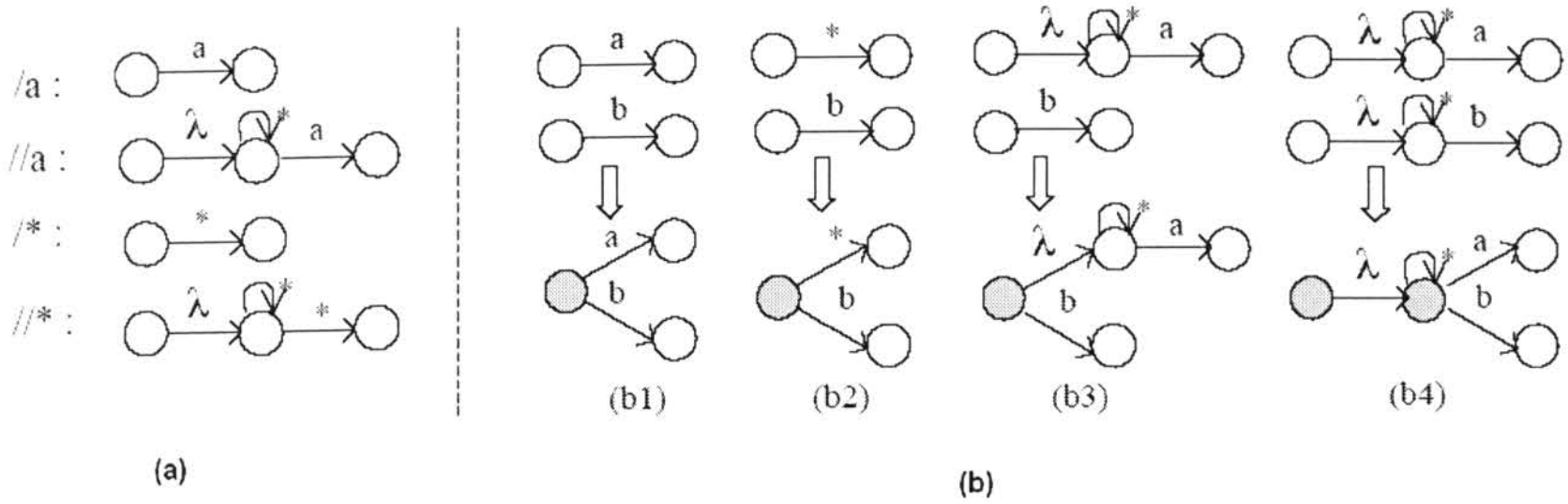
automaty spolu nijak nespolupracovaly, v YFilteru se ze všech dotazů dohromady postaví jediný *nedeterministický konečný automat* (NKA, Nondeterministic Finite Automaton). Největší výhoda je v tom, že společné prefixy dotazů jsou v NKA reprezentovány pouze jednou, což naprosto zásadně snižuje velikost (a paměťové nároky) stroje.

Na obrázku 5 je příklad NKA reprezentujícího osm dotazů (způsob konstrukce NKA vysvětlíme záhy). Kolečka odpovídají stavům, pokud je kolečko dvojité, jde o přijímací stav dotazu, jehož identifikátor je nad ním vyznačen, stínovaná kolečka odpovídají sdíleným stavům. Hrany s jednotlivými symboly (značící jednotlivé elementy) odpovídají přechodové funkci, přičemž symbol 'λ' značí přechod, který nevyžaduje vstup (tzv. λ-přechod), '*' nahrazuje jakýkoliv element.



Obr. 5: XPath dotazy (a) a jim odpovídající NKA (b)

Samotný způsob konstrukce NKA vidíme na obrázku 6. Začneme konstruováním jednotlivých NKA fragmentů (a), z nich zřetězením postavíme NKA odpovídající jednomu dotazu; tyto jednotlivé NKA zkombinujeme do většího NKA podle (b).



Obr. 6: Konstrukce NKA

Nyní popíšeme samotný proces filtrování. Jednotlivé přechody mezi stavy jsou řízeny událostmi generovanými XML parserem. Hnízdění XML dokumentů vyžaduje, aby v případě, že přijde událost "konec elementu", systém přešel do stavu, ve kterém byl před přijetím "začátek elementu". K tomuto potřebujeme využijeme zásobník, do kterého budeme zapisovat aktivní stavy. Jednotlivé stavy si o sobě musí pamatovat veškeré informace - ID, všechny přechody, zda je to koncový stav nebo "//-stav" (viz `//a` a `//*` na obrázku 6a) a pro koncové stavy také seznam příslušných dotazů. Samotná práce filtrovače probíhá podle přijaté události:

- *Začátek dokumentu*: Inicializujeme zásobník počátečním stavem.

- *Začátek elementu*: Když je z dokumentu čten nový element, NKA musí provést všechny odpovídající přechody - pro každý aktivní stav (tj. stav na vrcholu zásobníku) se musí vykonat následující:
  1. Vyhledáme jméno elementu mezi přechody z tohoto stavu. Pokud je přítomno, dáme tento nový stav či stavy do dočasné množiny.
  2. Také vyhledáme '*'. Pokud existuje přechod z našeho stavu, rovněž dáme nové stavy do dočasné množiny.
  3. Zkontrolujeme další informace o stavu. Pokud jde o "//-stav", přidáme do dočasné množiny i tento původní stav, čímž správně implementujeme tzv. self-přechod označený symbolem '*' na obrázku 6(a).
  4. Nakonec vykonáme případné λ-přechody z tohoto stavu tím, že vyhledáme 'λ', pokud je přítomno, pro příslušný "//-stav" rekurzivně vykonáme body 1 až 3.

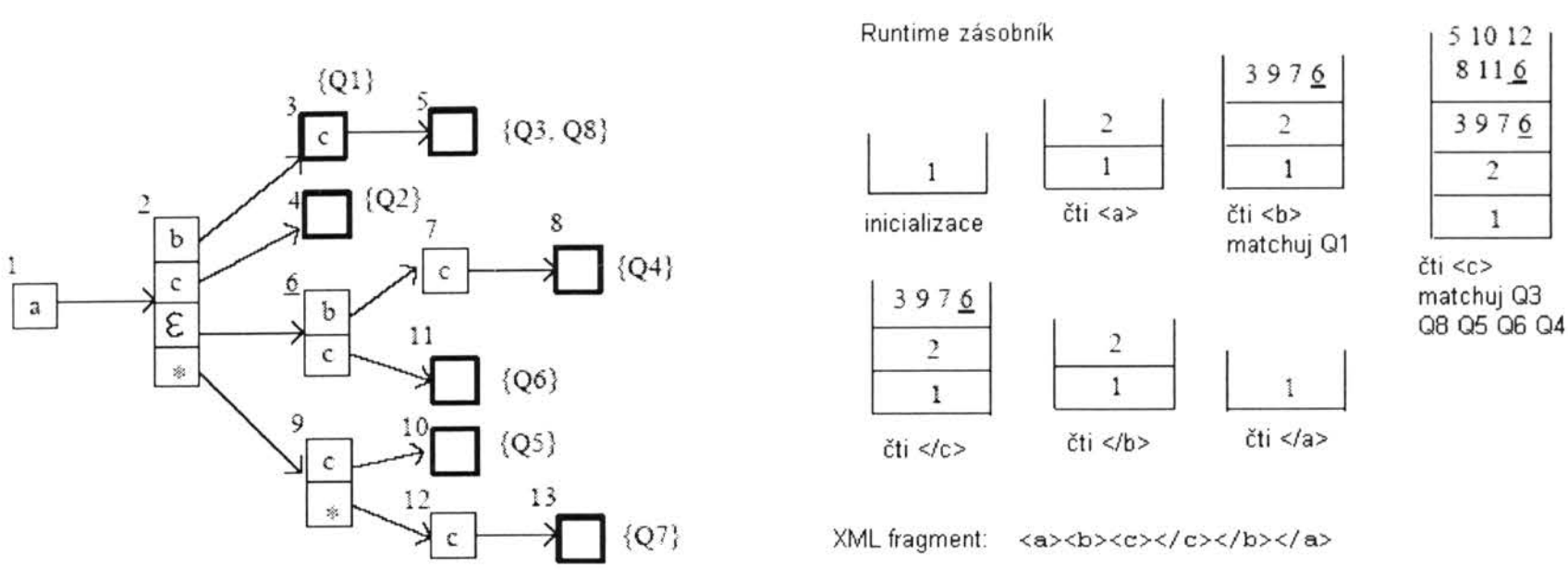
  Poté, co se tímto způsobem vypořádáme se všemi aktivními stavy, dáme dočasnou množinu na vrchol zásobníku. Tyto se tedy stanou aktivní pro příjem další události.

Pokud je některý ze stavů na vrcholu zásobníku koncovým stavem pro některý dotaz, přidáme příslušný dotaz (resp. jeho identifikátor) do výsledku.

- *Konec elementu*: Jednoduše odstraníme horní patro zásobníku.

Příklad běhu algoritmu je na obrázku 7. Vlevo máme NKA i se všemi informacemi, které si musí pamatovat, vpravo je zachycen vývoj zásobníku pro uvedený XML fragment. Pokud je stav v zásobníku podtržený, znamená to, že jde o "//-stav".

.



Obr. 7: Příklad běhu filtrovacího algoritmu

Je důležité si uvědomit, že zatímco tradiční NKA mívá obvykle za cíl najít jeden přijímací stav pro vstup, náš NKA musí pokračovat, aby mohly být dosaženy všechny potenciální konečné stavy. Důvod je ten, že musíme samozřejmě najít všechny dotazy, kterým dokument vyhovuje.

Základní implementace YFilteru podporuje pouze dopředné osy a testy na uzly. Tyto testy mohou být dvojího typu:

- Vnořené cesty se před počátkem filtrování "vybalí" a rozdělí na více poddotazů, YFilter je vyhodnocuje zvlášť a až na konci, jsou-li splněny všechny poddotazy, vydá zprávu o splnění celého dotazu. Příklad:

```
/zamestnanci/zamestnanec[deti]/jmeno
```

se rozdělí na následující dva poddotazy:

```
/zamestnanci/zamestnanec/deti
/zamestnanci/zamestnanec/jmeno
```

- Pro řešení *value-based* predikátů (ptáme se na shodu s konkrétní hodnotou) YFilter nabízí dvě strategie. První z nich je vyhodnocuje co nejdříve, tj. ihned poté, kdy jsou nalezeny příslušné elementy, které je nesou. Alternativní strategie naopak vyčkává, než se úspěšně vyhledá celá cesta odpovídající dotazu, a až poté aplikuje všechny příslušné predikáty. Experimenty ukázaly, že druhá strategie je většinou (snad navzdory intuici) úspěšnější.

5.2.3 Další systémy a přístupy

Zatímco většina filtrovacích algoritmů přímo zpracovává jednotlivé SAX události a dokument čte pouze jednou, někdy se vyplatí vstupní dokument aspoň trochu předzpracovat. IndexFilter, viz [3], je příkladem takového algoritmu. Dokument čte celkem dvakrát, při prvním čtení si o něm ukládá jisté (nepříliš obsáhlé) poznatky o dokumentu - vytváří tzv. index, do něhož si ukládá informace o hierarchii jednotlivých elementů. Po jeho úpravě pak čte dokument znovu, tentokrát již s pomocí indexu velmi efektivně filtruje. Pomocí indexu se vyhne profiltrovávání konkrétních úseků textu, které zaručeně nepřinese žádný výsledek. Pokud máme dlouhé dokumenty, tato technika vykazuje lepší výsledky než např. YFilter - vyplatí se efektivnější filtrování za cenu určitého zdržení při výrobě indexu.

V systému XTrie [6] se určité vybrané podřetězce původních XPath dotazů ukládají do speciálních indexových struktur - *trií* - a ty se poté vyhledávají v jednotlivých dokumentech jako celek. Hlavní myšlenkou tohoto přístupu je to, že výskyt celé konkrétní posloupnosti elementů je v dokumentu méně pravděpodobný než výskyt jednotlivých elementů.

Niagara [7] je filtrovací systém, který (narozdíl od všech dosud zmíněných) vyhledává primárně podle shody predikátů, což může být pro některé dokumenty a sady dotazů výhodnější. Dotazy se rozdělí do skupin tak, že v jedné skupině jsou dotazy obsahující test na tentýž atribut. Pokud by se dotaz dal zařadit do více skupin, speciální algoritmus se snaží vybrat co nejselektivnější test a podle tohoto pak dotaz zařadí do skupiny. Pokud se v

dokumentu najde atribut příslušející nějaké skupině, porovnají se hodnoty těchto atributů a dále postupují pouze dotazy, které se v hodnotě s dokumentem shodují.

5.3 Publish-subscribe systémy: architektura a konkrétní techniky

V kapitole 5.2 jsme se seznámili s metodami umožňujícími rychle a efektivně zjistit, kterým ze zaregistrovaných dotazů konkrétní dokument vyhovuje. Je ovšem také potřeba dokument správným příjemcům doručit. Pokud by měl každý poskytovatel na starosti doručování svých zpráv, musel by o příjemcích znát všechny potřebné informace týkající se jejich umístění a požadavků. Publish-subscribe systémy umožňují poskytovatele od těchto povinností zcela odstínit, díky tomu nemusí o případných příjemcích vědět vůbec nic. Vše zařizuje neutrální prostředník. Poskytovatel se na začátku zaregistruje u *registrační služby* a dál už bude pouze posílat dokumenty do systému tvořeného jedním brokerem či sítí brokerů; tento systém se již sám postará o samotné filtrování a doručování registrovaným uživatelům. Ani ti se nemusí o nic starat, pouze se předem zaregistrují a data jim poté chodí automaticky ze systému.

Nyní si postupně probereme nejdůležitější otázky spojené s XML publish-subscribe systémy - základní služby (kapitola 5.3.1), routování (5.3.2), seskupování uživatelských profilů pro účinnější paralelizaci (5.3.3) a pro přenos menší části původního XML dokumentu (5.3.4), nastíníme také možnost optimalizace množství přenášených dat (5.3.4).

5.3.1 Základní služby publish-subscribe systémů

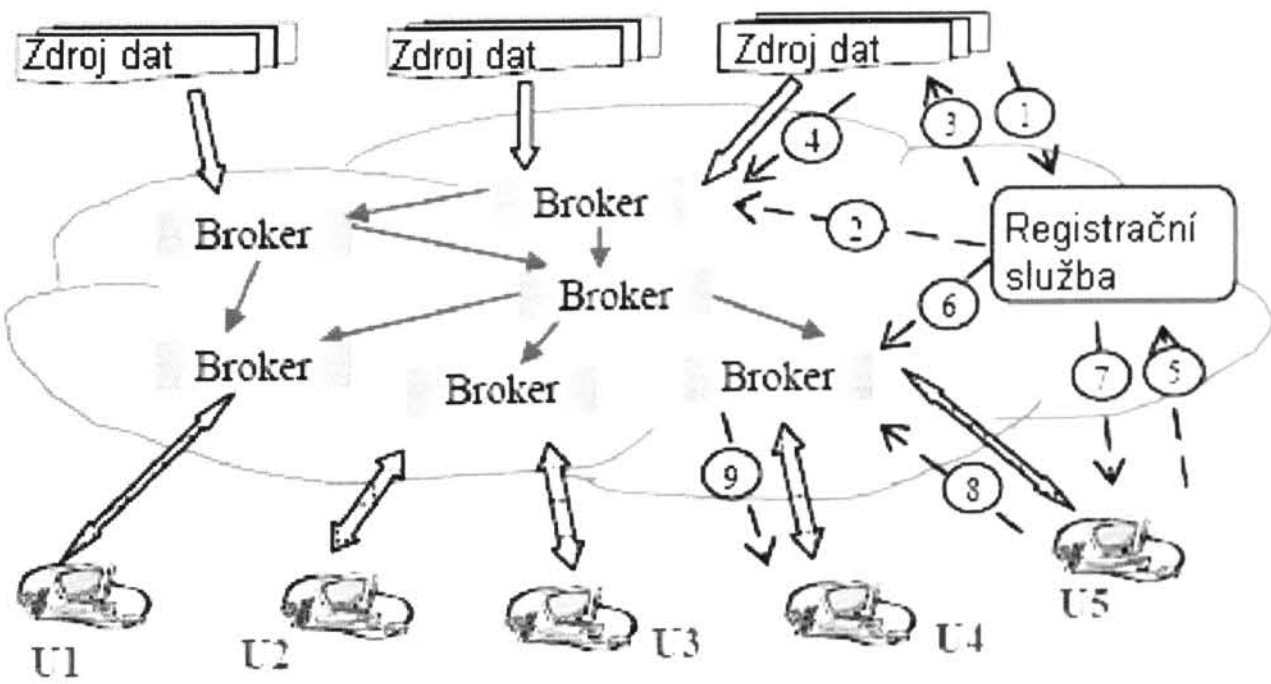
Nyní si s pomocí obrázku 8 probereme jednotlivé události, které v publish-subscribe systémech mohou nastat:

- *Registrace nového poskytovatele*: Datový zdroj kontaktuje registrační službu, poskytne jí o sobě informace týkající se svého umístění a případně i další informace jako XML schéma svých dokumentů či informace o tom, jak často bude zhruba posílat dokumenty do systému a jaká bude jejich očekávaná velikost (viz zpráva číslo 1 na obrázku 8). Registrační služba přidělí zdroji unikátní identifikátor a vybere pro něj *kořenový broker*. Volba kořenového brokeru pro konkrétní zdroj je v jednotlivých systémech implementována různě, závisí na topologické vzdálenosti brokeru od zdroje, momentálních možnostech sítě a také podle očekávaného objemu dat od

zdroje. Poté registrační služba přepošle informace o novém datovém zdroji kořenovému brokeru (zpráva 2) a nakonec pošle zdroji zpět ID a adresu brokeru, který mu přidělila (zpráva 3).

- *Poslání dokumentů do systému*: Poté, co se poskytovatel úspěšně zaregistroval, může vkládat data do systému - jednoduše je spolu se svým ID pošle na přidělený kořenový broker (zpráva 4).
- *Registrace nového uživatele*: K přihlášení do systému uživatel kontaktuje registrační službu a poskytne jí informace o svém umístění a svůj profil - dotaz či dotazy zapsané pomocí dotazovacího jazyka, obvykle XPath (zpráva 5). Registrační služba tomuto profilu přidělí identifikátor a vybere mu na základě umístění či obsahu profilu (viz kapitoly 5.3.3 a 5.3.4) *hostitelský broker*. Poté přepošle profil a informace o umístění uživatele hostitelskému brokeru (zpráva 7). Od této chvíle se bude hostitelský broker starat o veškeré záležitosti spojené s tímto profilem.
- *Aktualizace profilu*: Jakékoli následné požadavky na změnu profilu (a také jeho případné zrušení) již uživatelé posílají přímo na příslušný hostitelský broker.

Navíc bývají k dispozici metody, které využijí poskytovatelé zpráv, chtějí-li např. aktualizovat schéma svých dokumentů či jiné informace. Všimněme si, že uživatelé nepotřebují žádné metody pro přijetí zprávy, která vyhovuje jejich profilu, protože tyto zprávy jsou jim doručeny automaticky systémem (zpráva 9).



Obr. 8: Distribuovaný publish-subscribe systém

5.3.2 Routování

Nejjednodušší publish-subscribe systém sestává z jediného brokeru, do kterého se uloží uživatelské profily, poskytovatel na tento broker vkládá dokumenty a celé filtrování probíhá na tomto jediném brokeru. Někdy v takovém případě mluvíme o tzv. *centralizovaném řešení.* Máme-li ovšem brokerů více, můžeme i využít možnosti paralelizace - dotazy lze rozdělit do nezávislých skupin, vyhodnocování takovýchto skupin lze provádět paralelně ve stejném čase na různých brokerech, čímž uspoříme hodně času.

Předpokládejme pro teď, že máme dáno, jak konkrétně síť vypadá - tj. kolik brokerů máme k dispozici a jak mezi sebou vzájemně komunikují. Typicky nám tímto vzniká strom (resp. graf bez orientovaných cyklů), ve kterém každý z brokerů zná spojení k rodičovskému brokeru a ke všem svým synovským brokerům.

Pojďme se věnovat tomu, co se děje se zprávou, která je vložena do systému (konkrétně do některého z kořenových brokerů přiděleného zdroji registrační službou). V některých strategiích je jednoduše vždy rozeslána všem synům brokeru, toto se opakuje tak dlouho, dokud zpráva nedorazí na hostitelský broker, tam je pomocí některého z filtrovačů (viz kapitola 5.2) profiltrována a pokud vyhovuje některým z profilů sídlícím na tomto brokeru, je předána příslušným uživatelům. Nevýhodou tohoto principu je veliká redundance posílaných dat (strategii se trefně říká "flooding", čili záplava)- vnitřní brokery rozesílají všechno všude bez ohledu na to, že by věděly, jestli se vůbec někdo o příslušný dokument zajímá.

Routovací tabulky

Aby se zprávy posílaly efektivněji, je potřeba, aby každý broker v síti nějak zjistil, zda vůbec má cenu k němu doraziví dokument předávat dál (tj., jestli se o něj někdo "pod ním" zajímá) a pokud ano, tak kterému brokeru. K tomu si potřebuje v sobě pamatovat určité informace, které potřebuje k efektivnímu routování, obvykle v podobě routovací tabulky. V ní jsou zapsány routovací dotazy a k nim odpovídající linky na synovské brokery, které tyto dotazy zpracovávají. V rámci každého z brokerů funguje jedna instance nějakého filtrovače z minulé kapitoly, která v případě přijetí zprávy od poskytovatele nebo od rodičovského brokeru zajistí její profiltrování proti routovacím dotazům tohoto brokeru. Zpráva je pak předána synovským brokerům, případně přímo uživatelům, podle odpovídajících vstupů v routovací tabulce.

Přibližme si nyní způsob, jak realizovat routovací tabulky. Konstrukce začíná odspodu, tj. od hostitelského brokeru, na kterém "sídlí" skupina dotazů. Routovací tabulky hostitelských brokerů se skládají z párů {dotaz, odkaz na uživatele}. Ze všech dotazů z tohoto brokeru se dohromady vyrobí jeden dotaz, kterým bude tento broker zastoupen v tabulce rodičovského brokeru (tento "celkový" dotaz bude ve tvaru disjunkce jednotlivých poddotazů). V rodičovském brokeru se shromáždí routovací dotazy od všech synovských brokerů, udělá se z nich opět tabulka ve tvaru {routovací dotaz, odkaz na synovský broker} a z nich se opět vyrobí jeden celkový dotaz, který se propaguje nahoru k rodičovskému dotazu.

Pokud bychom postupovali přímo takto, nastal by patrně problém s velikostí routovacích tabulek. Budeme-li vždy zapisovat podrobně všechny poddotazy synovských brokerů, de facto to znamená, že musíme hned u kořenového brokeru přefiltrovat zprávu na všechny dotazy. Naší snahou je po cestě od kořenového k hostitelskému brokeru zprávu "odfiltrovávat postupně" a dotazy postupně zjemňovat. Proto se většinou zmíněný "celkový" dotaz nějakým způsobem zevšeobecní, než je proveden jeho zápis do routovací tabulky rodičovského brokeru.

Příklady konkrétních systémů

Ze systémů fungující na této bázi zmiňme ONYX [10] či XRoute [5]. Liší se zejména použitým filtrovačem - ONYX používá YFilter (kapitola 5.2.2) a XRoute používá XTrie (kapitola 5.2.3).

Zajímavý routovací protokol je implementován v systému SemCast [20]. Zatímco uvedené strategie (kromě strategie flooding z počátku kapitoly) vyžadovaly filtrování na každém brokeru, v SemCastu se filtruje pouze na kořenových a hostitelských brokerech. Ve chvíli, kdy zpráva dorazí do systému, je po přefiltrování poslána do nejméně jednoho konkrétního komunikačního kanálu a skrz něj dorazí až do hostitelského brokeru, kde je přefiltrována jemněji a předána příslušným uživatelům. Výhoda tohoto přístupu spočívá v tom, že celé routování ve vnitřních brokerech spočívá v pouhém přečtení identifikátoru kanálu, kterým má být zpráva posílána, a ihned je předána tímto kanálem o úroveň níž, čímž můžeme uspořit nemalou výpočetní kapacitu.

O to, jak budou v SemCastu organizovány jednotlivé kanály, se stará registrační služba. K tomu využívá jak syntax jednotlivých dotazů, tak i statistické informace, podle nichž jednou za čas síť kanálů přebuduje, aby se systém lépe přizpůsobil měnícím se podmínkám. Konkrétněji, nejprve se prozkoumá syntax dotazů, aby se s pomocí některého existujícího algoritmu (viz [24]) určily hierarchie mezi dotazy (tj. zda je některý dotaz nadmnožinou jiného) , poté na vzorku dokumentů zkoumá, jak moc se všechny profily vzájemně "překrývají", tj. jak velký poddíl zpráv vyhovuje konkrétní dvojici dotazů zároveň. Pomocí výpočetního modelu publikovanému v [20] jsou jednotlivé hierarchie a "téměř hierarchie" přiděleny jednotlivým kanálům a jsou reprezentovány dotazem, který v hierarchii stojí nejvýše.

Každý kořenový broker si poté bude držet ve své routovací tabulce informaci o routovacím dotazu a identifikátoru kanálu, kam se má dokument poslat v případě, že vyhovuje příslušnému dotazu. Vzhledem k tomu, že některé uzly sítě mohou sloužit jako brokery pro více než jeden kanál, může se stát, že broker dostane tutéž zprávu z více kanálů, a je v takovém případě zodpovědný za to, aby ji dál poslal pouze jednou.

5.3.3 Přidělování profilů na hostitelské brokery

Máme-li více brokerů, můžeme každému z nich přiřadit nějakou skupinu dotazů a tím zvýšit paralelizovatelnost systému, jak jsme již poznamenali v úvodu minulé podkapitoly. Dosud jsme se však nijak nezabývali tím, podle čeho registrační služba přiděluje jednotlivým profilům jejich hostitelský broker. Velice často toto služba rozhodne podle vzdálenosti uživatele od brokerů, tj. jednotlivé profily sídlí na nejbližších brokerech, či podle momentálního vytížení brokerů.

Můžeme však dotazy seskupovat i chytřeji. Zdá se být zřejmé, že pokud si dotazy sídlící na stejném hostitelském brokeru budou dostatečně podobné, můžeme filtrování zefektivnit. Zde ovšem narazíme na problém: pokud např. v jedné skupině budou dotazy s prefixem `/a/b` a v druhé `a/c`, tedy v podstatě "nepodobné" dotazy, stále se může stát, že dokument vyhovuje dotazům z obou skupin. Naším cílem je co největší "exkluzivita" dotazů ve skupině, čímž máme namysli, že splnění jedné skupiny dotazů pro konkrétní dokument zcela vylučuje splnění jiných skupin (tomuto cíli chceme alespoň přiblížit). Např. budeme-li mít skupinu

dotazů začínajících `/a/b[@id=1]` a skupinu dotazů začínajících `/a/b[@id=2]`, výrazně se sníží možnost, že by zpráva mohla splnit dotazy obou skupin. Zcela jistě je nemůže splnit, pokud je v dokumentu pouze jediný element `b`.

Je otázkou, jak dosáhnout takovéto exkluzivity. Dělení dotazů pouze podle struktury (bez predikátů) nestačí (viz první ze dvou příkladů z minulého odstavce). Pokud k tomu navíc uvažujeme i dělení podle predikátů, může to pomoci, pokud mají stejnou strukturu, obsahují predikát (testující hodnotu) na stejný cíl (tj. element nebo atribut), používají operátor '=', ale porovnávají s rozdílnou hodnotou. Ani v tomto případě to ještě nemusí být jednoznačné - pokud se v dokumentu elementy opakují, je exkluzivita opět limitována. Dalším problémem je fakt, že podle kritéria budeme pravděpodobně schopni rozdělit jen poměrně malou část dotazů, neboť zájmy uživatelů bývají různorodé - většina dotazů by pak spadla do *směsné skupiny* seskupující takové dotazy, které kritérium vůbec neobsahují, a výhody by se tudíž nevyužily. Tento poslední problém lze řešit tím, že kritérií bude více.

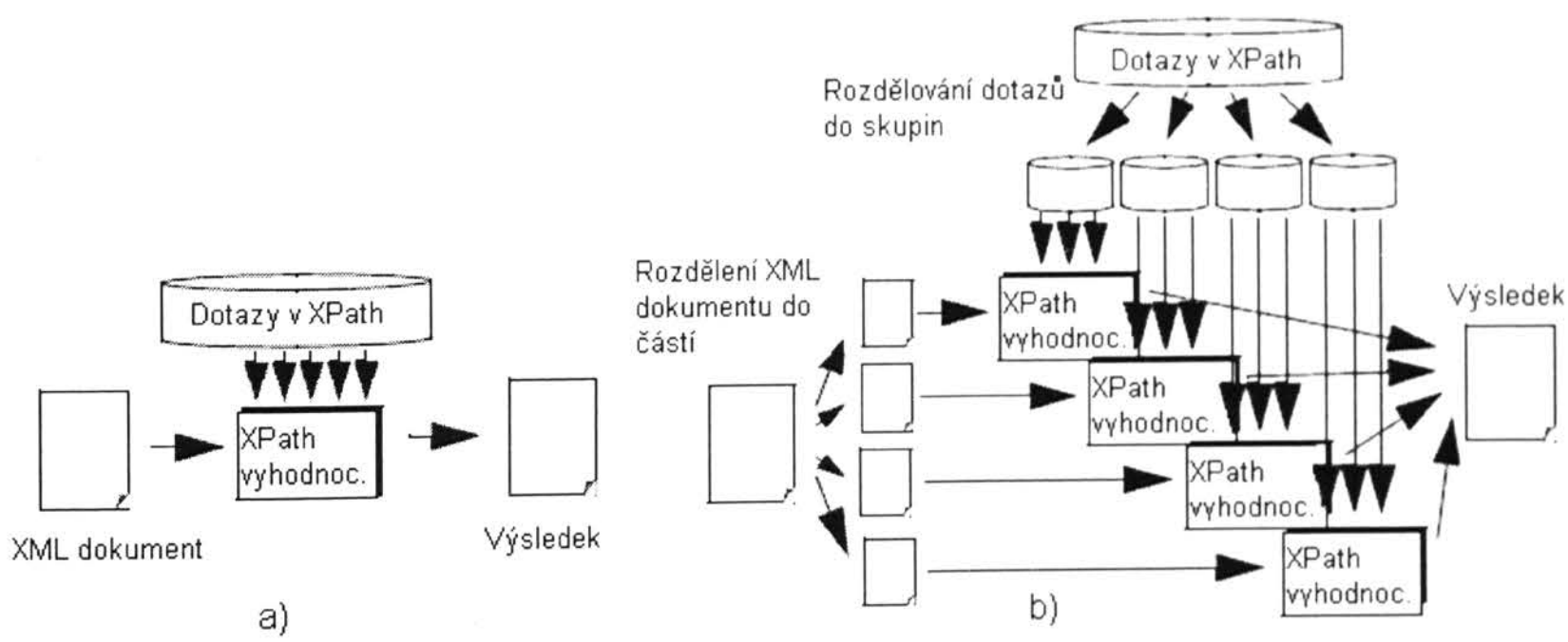
Jeden konkrétní algoritmus pro výběr "téměř exkluzivních" kritérií je nastíněn v [10]. Experimenty prokázaly, že algoritmus vhodným sdílením skutečně snižuje zatížení sítě. Nyní si jej krátce přiblížíme.

1. Nejprve ze všech dotazů vygenerujeme množinu všech možných predikátů - kritérií, tu setřídíme sestupně podle toho, kolik dotazů z naší množiny predikát obsahuje. Poté hladovým algoritmem vybereme podmnožinu z nich tak, že v každém dotazu je obsažen alespoň jeden predikát z této podmnožiny.
2. Po předchozím kroku máme vybráno M predikátů. Z nich se nyní snažíme vytvořit K skupin. U každého z predikátů rozdělíme obor hodnot do K přihrádek. Poté se do získaných K*M přihrádek přidělí jednotlivé dotazy podle toho, jakou hodnotu v příslušném kritériu obsahují. Vzhledem k tomu, že v každém dotazu je obsažen alespoň jeden z M predikátů, musí každý dotaz patřit minimálně do jedné přihrádky. Pokud lze dotaz dát do více přihrádek, vybereme mu z nich jednu náhodně.
3. K skupin dotazů poté získáme tím, že přidělíme dotazy i-té přihrádky každé z M sad i-té skupině dotazů.

Další techniku si představíme níže v kapitole 5.3.4. S touto technikou můžeme dosáhnout naprosté exkluzivity, což nám kromě zrychlení filtrování navíc umožní přenášet menší objem dat.

5.3.4 Přenos menší části XML dokumentu

Jak jsme již poznamenali v kapitole 5.3.3, je výhodné dotazy neseskupovat na hostitelské brokery zcela náhodně, ale podle nějaké míry podobnosti – pokud dotazy rozdělíme chytře, můžeme nejen snížit zatížení sítě, ale můžeme i dokonce docílit toho, že na vyhodnocení dotazu ve skupině nemusíme na broker posílat celý dokument, ale jen jeho určitou část, která může být řádově o mnoho menší než původní dokument. Je zřejmé, že pokud aplikujeme dotazy na menší část dokumentu, filtrování se urychlí. Tuto zajímavou techniku si nyní přiblížíme trochu důkladněji. Další informace o této metodě lze najít v [17].



Obr. 9: Filtrování dokumentu vcelku (a) a po fragmentech (b)

Příprava dotazů

Pro tuto konkrétní techniku potřebujeme znát schéma XML dokumentů, nad kterým systém pracuje. V první fázi využijeme znalosti schématu předávaných zpráv k tomu, že si jednotlivé XPath dotazy „vyčistíme" a přetransformujeme. (Je důležité si uvědomit, že tato přípravná fáze neovlivní časově kritickou filtrovací fázi, která zahrnuje rozdělení dokumentu do jednotlivých částí a samotnou filtraci.) V této fázi převedeme veškeré XPath dotazy (ve

kterých opět povolíme pouze dopředné osy) do určité standardizované formy – do takového tvaru, kde nebude osa `descendant::` a kde nebudou užity masky.

Tento krok, ve kterém budeme nahrazovat osu `descendant::` a masky, úzce souvisí s definicí schématu zpráv. Pomocí něj osu `descendant::` či masku rozvineme na všechny přípustné možnosti. Musíme si však dávat pozor na to, že dokument XML může teoreticky mít neomezenou hloubku, pokud příslušné XML schéma povoluje rekurzi mezi elementy.

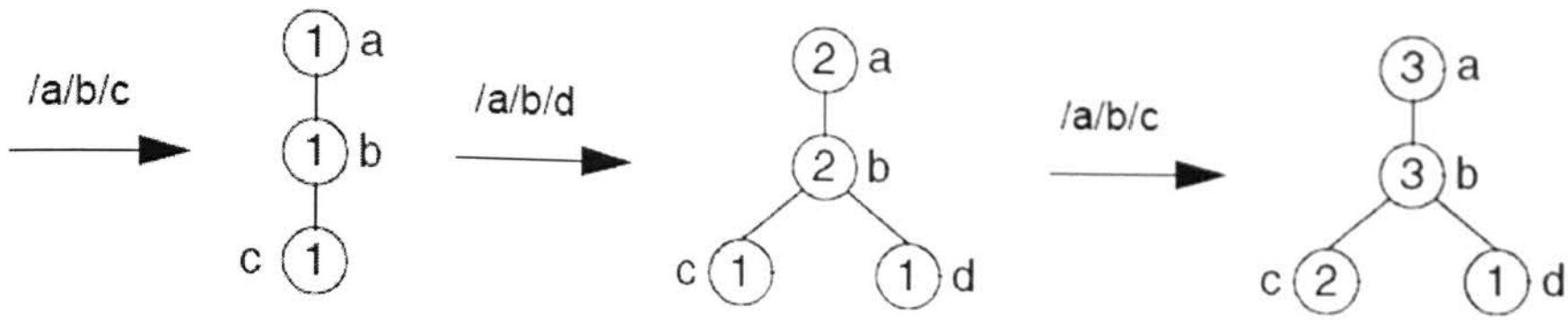
Navíc rovnou vyřadíme takové XPath dotazy, u kterých je po pouhém porovnání se schématem jisté, že nemohou vést k žádnému řešení.

Popis algoritmu a konkrétní příklad

Nyní popíšeme algoritmus vytvářející *rozpadové schéma*, kterým tuto upravenou sadu XPath dotazů rozdělíme do skupin, s tím, že dotazy v jedné skupině budou mít stejný prefix. Tento prefix nám pak bude určovat podstrom původního XML dokumentu, který bude pro skupinu relevantní; na hostitelský broker této skupiny dotazů tedy stačí posílat pouze tuto část dokumentu namísto dokumentu celého. Nejprve si musíme určit, do kolika skupin je chceme dělit – toto rozhodneme podle toho, kolik máme k dispozici brokerů a kolik dotazů v systému očekáváme.

Dotazy se snažíme do skupin rozdělit "spravedlivě", tj. tak, aby všechny skupiny byly pokud možno stejně velké. Avšak vzhledem k tomu, že se samozřejmě může objevit dotaz, který nemůžeme zařadit do žádné z vytvořených skupin (a to proto, že s žádnou z nich nesdílí prefix), založíme navíc ještě jednu skupinu, *směsnou skupinu* (potpourri set). Tato skupina by měla být co možná nejmenší, neboť ji musíme vždy vyhodnocovat proti celému dokumentu.

Nejprve si vytvoříme vážený prefixový strom všech dotazů, tj. strom, kde uzlům odpovídají jednotlivé elementy dotazu a hranám odpovídá vztah rodič-potomek; váha každého z uzlů stromu je na začátku rovna jedné a zvyšuje se pokaždé, kdy do stromu přidáváme další dotaz, který tento uzel zahrnuje. Nejlépe si to ukážeme na příkladu z následujícího obrázku, ve kterém postupně přidáváním dotazů `a/b/c`, `/a/b/d`, `/a/b/c` tvoříme prefixový strom.



Obr. 10: Tvorba prefixového stromu

Nyní se dotazy pokusme vhodně rozdělit do (v tomto případě dvou) skupin. Naivní způsob, jak toho dosáhnout, by bylo setřídění uzlů podle jejich váhy a použití dvou uzlů s největší váhou jako tzv. bodů rozpadu, tyto body nám budou definovat jednotlivé skupiny. V našem příkladu by se jednalo o `/a` a `/a/b`. Bohužel ovšem `/a` reprezentuje kořenový uzel, první skupina by tudíž musela operovat na celém dokumentu. Navíc i `/a/b` může vzhledem ke konkrétnímu XML schématu reprezentovat (víceméně) celý dokument, takže ani zde se nedočkáme zmenšení dokumentu, se kterým skupina pracuje. Přitom je evidentní, že optimální řešení spočívá v rozdělení na skupiny definované cestami `/a/b/c` a `/a/b/d`.

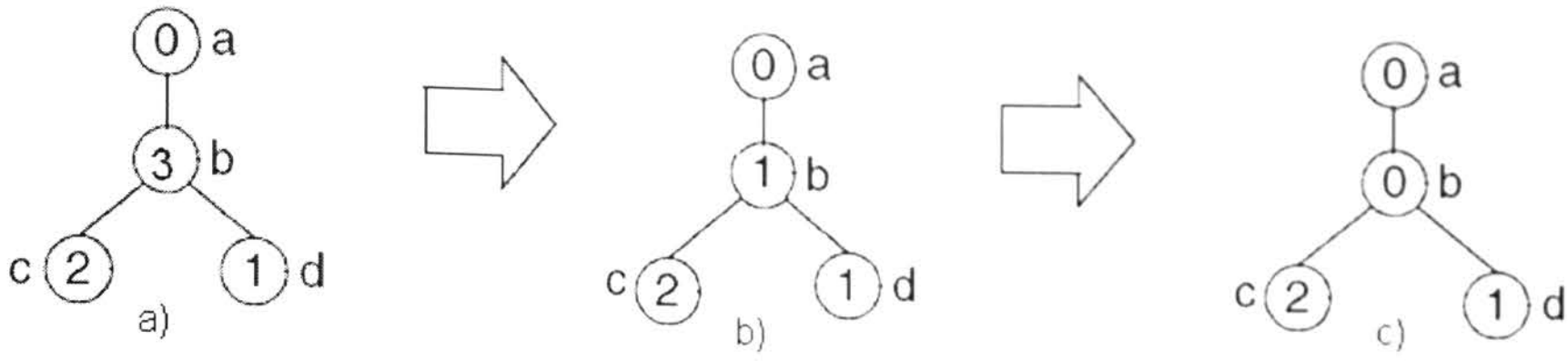
Ukážeme si účinnější strategii. Nejprve uvedeme pseudokód popisující algoritmus, poté si algoritmus vysvětlíme na příkladu.

```
Algoritmus VygenerujRozpadovéSchéma
Input:  D //kořen prefixového stromu dotazů
        C //na kolik skupin chceme dělit (počet bodů rozpadu)
Output: S //uzly prefixového stromu reprezentující skupiny

BEGIN
  S=∅ //výsledková množina
  DO
    N=traverse(D)-{Root(D)} //všechny uzly kromě kořenu
    u_max=0

    //nyní hledáme nejtěžší uzel, který ještě není
    //ve výsledkové množině
    FOREACH u∈(N-S)
      IF (váha(u)>váha(u_max)
        u_max:=u
      END IF
    END FOREACH

    //dokud má syn stejnou váhu, vybírej jeho
    p=u_max;
    WHILE EXISTS (c;  ( c∈(children(p)) AND (váha_c = u_max)))
```

```
      u_max = c
      p = c
    END WHILE

    //odečteme váhu u_max od předků
    FOREACH p∈(předek(u_max))
      váha(p)=váha(p)-váha(u_max)
      IF (váha(p) <0)
        váha(p)=0
      END IF
    END FOREACH

    //odstraníme z výsl. množiny lehčí uzly
    FOREACH e∈S
      IF (váha(e)<váha(u_max))
        S = S-{e}
      END IF
    END FOREACH
    S = S ∪ {u_max}

  //toto dělej, dokud nemáme všechny body rozpadu
  WHILE (|S|< C OR |S| ≠ |N|)

  //vrať výsledkovou množinu
  RETURN (S)
END
```

Hned na začátku vyloučíme kořenový uzel z množiny možných bodů rozpadu. Dále vyhledáme uzel s největší váhou, který je zároveň co nejhlouběji. Toho dosáhneme tím, že pokračujeme od uzlu s nejvyšší váhou dále k jeho synům, dokud mají stejnou nejvyšší váhu. Tímto způsobem získáme co nejdelší prefixy, které budou ve výsledku znamenat menší dokumentové fragmenty. Výsledný nejtěžší uzel přidáme jako do množiny řešení. Aby v další iteraci algoritmus opět nevybral tento stejný prefix (nebo část z něj), vypočítáme nové váhy uzlů po cestě od tohoto vybraného uzlu vzhůru ke kořenu; novou váhu získáme tak, že od staré váhy odečteme hodnotu váhy právě vybraného uzlu (pokud by nová váha některého z uzlů měla klesnout pod nulu, položíme ji rovnu nule). Váha právě vybraného uzlu se však nepřepočítává a zůstává na své původní hodnotě.

Vrátíme-li se k prefixovému stromu z obrázku 10, vybrali bychom v první iteraci uzel `/a/b` a přidáme ho do množiny výsledku. Na obrázku 11a) máme zachycenu situaci poté, co snížíme váhy rodičovských uzlů.



Obr. 11: Příklad běhu algoritmu

Další iterace vybere `/a/b/c` do množiny výsledku, protože `/a/b` teď vybírat nesmíme (protože je momentálně ve výsledkové množině) a `/a/b/c` má ze zbývajících uzlů nejvyšší váhu. Poté odečteme jeho váhu od všech předků (přičemž váhu uzlu `/a` už snížit nemůžeme, neboť je již rovna nule). Viz obrázek 11b).

Tato operace však přináší překvapení: Uzel `/a/b`, který jsme dříve vybrali do výsledkové množiny, má momentálně nižší váhu než naposledy přidávaný uzel `/a/b/c`, a proto musí být z výsledkové množiny vyřazen.

V další iteraci vybereme do výsledkové množiny uzel `/a/b/d`, protože má nejvyšší váhu (ve srovnání s `/a/b`). Opět musíme ještě aktualizovat váhy předků (obrázek 11c). V tuto chvíli algoritmus pro náš strom dotazů končí, neboť ve výsledkové množině máme právě dva uzly – *body rozpadu*, což bylo naším cílem.

Tím se nám podařilo původní skupinu dotazů rozdělit na dvě menší skupiny, z nichž každou lze řešit nezávisle na druhé na oddělených brokerech a na dokumentových fragmentech, které jsou určeny vypočítanými body rozpadu. V našem případě nebylo třeba použít směsnou skupinu, protože prefixy všech dotazů v prefixovém stromu byly pokryty některým z bodů rozpadu. V typickém případě však musíme směsnou skupinu založit, tvořit ji budou dotazy, které do ostatních skupin nelze zařadit.

5.3.5 Optimalizace přenosu XML dat

Přenos XML dat lze dále zefektivňovat, a to jak z hlediska množství přenášených dat, tak také z hlediska počtu provádění nutných přidružených operací. Tyto optimalizace lze mezi sebou samozřejmě kombinovat.

Komprese

XML je velice "upovídaný" jazyk. Objem dat lze snížit některým z kompresních algoritmů, např. XMill nebo ZLIB (viz [18]). Daní za toto snížení objemu je nutnost dekomprimovat po vstupu do každého brokeru, na kterém je prováděna filtrace.

Změna kódování

XML dokumenty musí být na každém brokeru pomocí parseru analyzovány a rozebrány na jednotlivé události, což může být časově velice náročné. Můžeme tomu však čelit tím, že dokument analyzujeme pouze na prvním brokeru a poté tato již zanalyzovaná data posíláme v nějakém vhodné reprezentaci (obvykle binární) tak, aby se na ně v synovském brokeru mohl rovnou spustit filtrovač bez nutnosti data parserem analyzovat znovu.

6 XmlPart - prototypová implementace XML publish-subscribe systému

Vytvořili jsme vlastní publish-subscribe systém XmlPart, na kterém můžeme demonstrovat rozmanitost technik používaných v publish-subscribe systémech. Z tohoto důvodu byl systém XmlPart navržen tak, aby umožňoval jednoduše měnit použitou strategii (konkrétně nastavením parametru při startu) a aby také bylo možné do systému přímočarým způsobem přidávat nové strategie.

Pro vlastní systém jsme implementovali několik variant různých strategií, vyzkoušeli jsme je na reálných datech a porovnali jsme je vzhledem k různým kritériím (čas potřebný na filtrování dokumentu, množství dat vyměněných mezi jednotlivými komponentami systému, počet brokerů v systému).

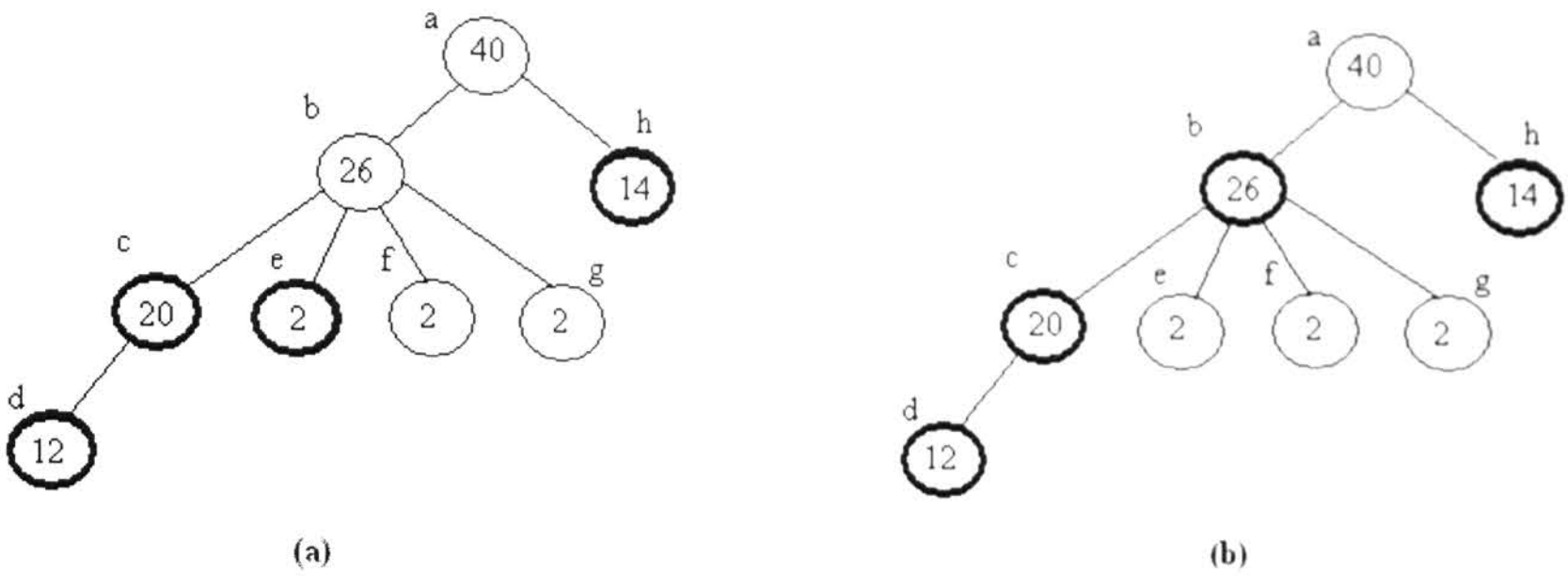
6.1 Nový způsob generování rozpadového schématu

Pro účely této práce jsme navrhli strategii, která vychází z techniky popsané v kapitole 5.3.4, přičemž poskytuje o něco lepší výsledky.

Podobně jako strategie z kapitoly 5.3.4, i zde musíme jednotlivé dotazy upravit (rozvineme `//` a `*` v dotazech pomocí zaregistrovaného DTD), poté z nich podobným způsobem generujeme rozpadové schéma z váženého prefixového stromu dotazů (tvorba tohoto stromu viz kapitola 5.3.4), které nám umožní na jednotlivé hostitelské brokery seskupovat dotazy tak, že na tyto brokery stačí posílat pouze relevantní části (podstromy) dokumentu, čímž ušetříme kapacitu sítě (a tedy i rychlost přenosu dat mezi jednotlivými brokery).

Podívejme se na vážený prefixový strom dotazů na obrázku 12(a). Zvýrazněné uzly značí body rozpadu, jak nám je vypočítal algoritmus `VygenerujRozpadovéSchéma` z kapitoly 5.3.4. Vidíme, že dotazy `/a/b/f` či `/a/b/g` musíme zařadit do směsné skupiny a budeme ji muset vyhodnocovat oproti celému dokumentu. Naším cílem je mít jednotlivé skupiny co

největší a zároveň co nejvyváženější, naopak směsnou skupinu se snažíme udržovat co nejmenší. Srovnejme body rozpadu z obrázku 12(a) s těmi na obrázku 12(b). Druhé schéma je vyváženější (váha nejmenší skupiny je maximální) a podařilo se nám zmenšit (v tomto konkrétním případě dokonce zcela potlačit) směsnou skupinu.



Obr. 12: Body rozpadu prefixového stromu podle strategie z kap. 5.3.4 (a) a podle naší nové strategie (b)

6.1.1 Algoritmus

Nyní uvedeme celý algoritmus v pseudokódu. Nejprve definujeme procedury `Označ` a `Odznač`, které mají na starosti přidávání resp. odebírání uzlů z výsledkové množiny, přitom zároveň přepočítávají váhy předků. Po těchto pomocných procedurách následuje samotný algoritmus generující nové rozpadové schéma, přičemž tento algoritmus využívá pomocné procedury.

```
Procedura Označ
Input: v //uzel, který vkládáme do výsl. množiny

//procedura dá uzel v do výsledkové množiny a přepočítá váhy
//předků
BEGIN

  //vrchol přidáme do výsledkové množiny
  S = S ∪ {v}

  //postupujeme od v směrem ke kořeni, uzlům po cestě
  //snížíme váhu o váhu(v)
  k = v
  DO
```

```
    k = otec(k)
    váha(k) = váha(k) - váha(v)

  //toto vykonáváme pouze do chvíle, kdy zjistíme, že uzel,
  //jemuž jsme naposledy snižovali váhu, je ve výsledkové
  // množině
  WHILE ( (k ∉ S) AND (k ≠ Root(D)) )
END
```

```
Procedura Odznač
Input: v //uzel, který vyndaváme z výsl. množiny

//procedura vyjme uzel z do výsledkové množiny a přepočítá
//váhy předků
BEGIN

  //vrchol odebereme z výsledkové množiny
  S = S - {v}

  //postupujeme od v směrem ke kořeni, uzlům po cestě
  //zvýšíme váhu o váhu(v)

  k=v
  DO
    k=otec(k)
    váha(k)=váha(k)+váha(v)

  //toto vykonáváme pouze do chvíle, kdy zjistíme, že uzel,
  //jemuž jsme naposledy zvyšovali váhu, je ve výsledkové
  // množině
  WHILE ( (p ∉ S) AND (p ≠ Root(D)) )
END
```

```
Algoritmus VygenerujRozpadovéSchéma2
Input:  D //kořen prefixového stromu dotazů
        C //na kolik skupin chceme dělit (počet bodů rozpadu)
Output: S //uzly prefixového stromu reprezentující skupiny

BEGIN
  S=∅ //výsledková množina
  DO
    N=traverse(D)-{Root(D)} //všechny uzly kromě kořenu
    u_max=0

    //nyní hledáme nejtěžší uzel, který ještě není
    //ve výsledkové množině
```

```
FOREACH u∈(N-S)
  IF (váha(u)>váha(u_max)
    u_max:=u
  END IF
END FOREACH
IF (u_max = 0)
  break
END IF

//dokud má syn stejnou váhu, vybíráme jeho
p=u_max;
WHILE EXISTS (c;  ( c∈(children(p)) AND (váha_c = u_max)))
  u_max = c
  p = c
END WHILE

//vybraný uzel označíme
označ(u_max)

//postupně odznačíme předky u_max, které mají menší váhu
//než u_max
p = parent(u_max)
WHILE ( p ≠ Root(D))
  IF ( (p∈S) AND (váha(p) < váha(u_max)) )
    odznač(p)
  END IF
    p=parent(p)
  END WHILE

  //cyklus provádíme tak dlouho, pokud zbývají skupiny ke
  //generovani
  WHILE ( (|S|<C) AND (|S|≠|N|) )

  //vrátíme výsledkovou množinu
  RETURN (S)
END
```

6.1.2 Správnost algoritmu

Narozdíl od původního algoritmu z kapitoly 5.3.4 lze o tomto novém dokázat, že poskytuje skutečně dobré rozdělení, i když pravděpodobně stále ještě ne optimální (bylo by nutné uvážit velikosti částí XML dokumentů odpovídajících jednotlivým podstromům - chtělo by to kromě DTD i statistiku XML dokumentů).

Ukážeme, že náš algoritmus poskytuje vyvážené rozpadové schéma, tj. takové, že váha nejlehčí skupiny je maximální. Postupujeme sporem. Nechť tedy náš algoritmus najde schéma S_1 s k skupinami s minimální váhou skupiny m_1 a nechť pro spor existuje schéma S_2 (jeho body rozpadu označme w_1, ..., w_k) s minimální váhou skupiny m_2 tž. $m_2 > m_1$. Pak ve schématu S_1 existuje bod rozpadu v takový, že váha jeho skupiny je menší než m_2. Uvažme situaci ve chvíli, kdy náš algoritmus přidá uzel v do S_1. Vzhledem k tomu, že váha jeho skupiny je menší než m_2, musí platit, že v každé ze skupin v S_2 musí mít mezi algoritmem vybranými body rozpadu do S_1 po jednom uzlu ze skupin, kterým v S_2 odpovídají uzly w_1, ..., w_k (jinak by totiž algoritmus musel místo v vybrat některý z uzlů w_1, ..., w_k). To ovšem znamená, že by po přidání uzlu v bylo vybráno k+1 uzlů, což je spor (algoritmus vybere vždy k vrcholů).

Vzhledem k tomu, že v jedné iteraci přidáváme do výsledkové množiny právě jeden uzel a zároveň v téže iteraci může být i více uzlů z výsledkové množiny naopak vyjmuto, nabízí se otázka, zda se algoritmus vždy zastaví (zastavovací podmínkou algoritmu je totiž dosáhnutí předem dané mocnosti výsledkové množiny nebo stav, kdy už jdou všechny vrcholy přidány). Kladnou odpověď přináší následující důkaz:

Budeme opět postupovat sporem. Nechť je tedy algoritmus nekonečný a nechť P je nekonečná posloupnost přidávání a odebírání jednotlivých uzlů z výsledkové množiny dokazující nekonečnost algoritmu. Vezměme takový uzel, který se v posloupnosti vyskytuje nekonečně mnohokrát a zároveň je v prefixovém stromu dotazů co nejhlouběji; je-li jich více, vezmeme libovolný z nich. Nechť je to uzel u. Je tam nekonečně mnohokrát, tj. je do výsledkové množiny nekonečně mnohokrát dáván a nekonečně mnohokrát je z ní tudíž i vyjímán. Vyjmut může být pouze poté, kdy jsme do množiny vložili jeho těžšího potomka. Alespoň jeden z jeho potomků tedy zavinil nekonečně mnoho vyjmutí uzlu u, nechť je to potomek p. Tento uzel p se tudíž také musel v P objevit nekonečně mnohokrát. Vzhledem k tomu, že p je potomkem u, musí se v prefixovém stromě nacházet hlouběji než u, což je spor s předpokladem, že jsme uzel u vybrali jako nejhlubší z těch, které se v P objevují nekonečně mnohokrát.

6.1.3 Vnořené cesty

Dosud jsme předpokládali, že dotazy jsou ve tvaru jednoduchých cest, z těchto cest jsme tvořili prefixový strom dotazu a potažmo rozpadové schéma. Měli bychom si však umět poradit i s dotazy, které v sobě obsahují vnořené cesty, např. `/a/b[c/d]/e/f`. Navrhli jsme dvě možná řešení tohoto problému:

Přímočarý a jednoduchý přístup k tomuto problému je uvažovat z dotazu pro tvorbu rozpadového schématu pouze část před první vnořenou cestou, pro dotaz z našeho příkladu by šlo o prefix `/a/b`. Nedostatek tohoto řešení je v tom, že jednotlivé body rozpadu poté vyjdou "zbytečně mělké", budou mít kratší prefix, než by mohly mít (a budou se tudíž na synovské brokery přenášet větší podstromy původního XML dokumentu).

V případě, že by se nám podařilo vnořenou cestu vyřešit jinak, mohli bychom z našeho dotazu pro tvorbu schématu použít `/a/b/e/f`, což by mohlo vést k "hlubším" bodům rozpadu a důsledkem toho by se skupina příslušející k tomuto bodu rozpadu mohla vyhodnocovat na mnohem menší části původního dokumentu. Navrhli jsme řešení: tvoříme schéma z dotazů tak, že z nich případnou vnořenou cestu vyjmeme (v našem případě bychom tedy pro tvorbu rozpadového schématu uvažovali `/a/b/e/f`) s tím, že je třeba si pamatovat, že ve chvíli, kdy se na synovský broker posílá část dokumentu definovaná příslušným bodem rozpadu, musí se přidat i části dokumentu definované vnořenou cestou.

6.2 Návrh systému

Systém XmlPart se skládá ze čtyř aplikací, které odpovídají jednotlivým součástem publish-subscribe systému, viz obrázek 8. Jsou to:

- *Koordinátor* – "srdce" systému – stará se o registraci dalších komponent a nastavuje routovací tabulky na jednotlivých brokerech. Koordinátor (na rozdíl od ostatních komponent) je v celém systému jen jeden.
- *Broker* – základní stavební jednotka doručovací služby. Každému brokeru je koordinátorem nastavena routovací tabulka, na základě které broker přepošle přijatý dokument (nebo jeho část) dalšímu brokeru nebo příjemci zpráv.
- *Poskytovatel zpráv*

- *Příjemce zpráv*

Všechny tyto aplikace existují ve dvojí verzi: GUI aplikace pro vizualizaci toho, co se v systému děje, a jako dávkové (command-line) pro testování a měření výkonnosti.

6.3 Zjednodušení použitá při implementaci

Na rozdíl od reálných systémů, které umožňují registrovat poskytovatele dat s různými schématy XML zpráv, používá XmlPart schéma jediné – zadává se ve formě DTD souboru jako první parametr při spouštění aplikace koordinátor. Z pohledu prototypové aplikace zaměřené na testování se samozřejmě nejedná o žádné vážné omezení (testy zpravidla provádíme nad jedním typem dokumentu), v případě implementace produkčního systému by však bylo nutné toto omezení odstranit.

Do doručovací služby zajišťované systémem XmlPart lze zapojit libobovolné množství brokerů, přičemž výsledná síť bude vždy "dvoupatrová", tj. budeme mít jeden kořenový broker a ostatní brokery budou sloužit jako hostitelské pro své skupiny dotazů. Topologie sítě tvořené brokery a komunikačními kanály tedy netvoří obecný strom.

Další omezení plyne z použití filtrovacího systému YFilter jako knihovny používané na brokerech – zde jsme místo implementace vlastního fitrovacího systému použili již hotové řešení, protože vývoj vlastního filtrovače by jen zvětšoval množství řádek programu, který se má navíc zabývat spíše technikami pro distribuované systémy než technikami pro tvorbu knihoven na filtrování XML dokumentů. YFilter je podobně jako XmlPart pouze prototypová implementace, takže kromě zakázání zpětných os (viz kapitola 5.1) zakazuje i některé další prvky syntaxe jazyka XPath. Neumožňuje například specifikovat osy pomocí dvojité dvojtečky(::), uvádět řetězcové literály v apostrofech, také povoluje jen jednu úroveň vnoření cest. Konkrétní gramatika přijímaná systémem YFilter (a tedy i systémem XmlPart) je uvedena v Dodatku A.

6.4 Instalace systému XmlPart

Pro úspěšnou instalaci je potřeba mít na počítači nainstalovanou Javu 5.0 (dá se stáhnout z http://java.sun.com a verze pro MS Windows je nahrána i na CD se systémem XmlPart) a mít správně nastavenou proměnnou PATH k programu java.exe, aby se po zadání příkazu `java -version` vypsalo na obrazovku něco podobného následujícímu výpisu:

```
C:\WINDOWS\Plocha>java -version
java version "1.5.0_03"
Java(TM) 2 Runtime Environment, Standard Edition (build
1.5.0_03-b07)
Java HotSpot(TM) Client VM (build 1.5.0_03-b07, mixed mode)
```

Vlastní instalace systému je pak velmi jednoduchá – stačí na libovolné místo na disku překopírovat (dále předpokládejme, že jsme kopírovali do adresáře `C:\`) adresář `program`. Doporučujeme rovněž tamtéž zkopírovat adresář `testovaci-data`, ve kterém jsou uloženy složky s testovacími soubory, příslušné DTD a sada dotazů. Popis spouštění a ovládání jednotlivých aplikací je uveden v následující podkapitole.

6.5 Ovládání systému XmlPart

Jak již bylo zmíněno dříve, systém XmlPart se skládá ze čtyř aplikací. Tyto aplikace mohou být spouštěny víceméně v libovolném pořadí, výjimku však tvoří koordinátor, který může být na jednom počítači spuštěn jen jednou a musí být navíc spuštěn dříve než ostatní programy realizující zbývající komponenty systému. Ostatní komponenty lze zapojovat a vypojovat v libovolném pořadí, s tím, že jednotlivým uživatelům budou samozřejmě doručeny pouze zprávy, které byly do systému publikovány až po registraci těchto uživatelů.

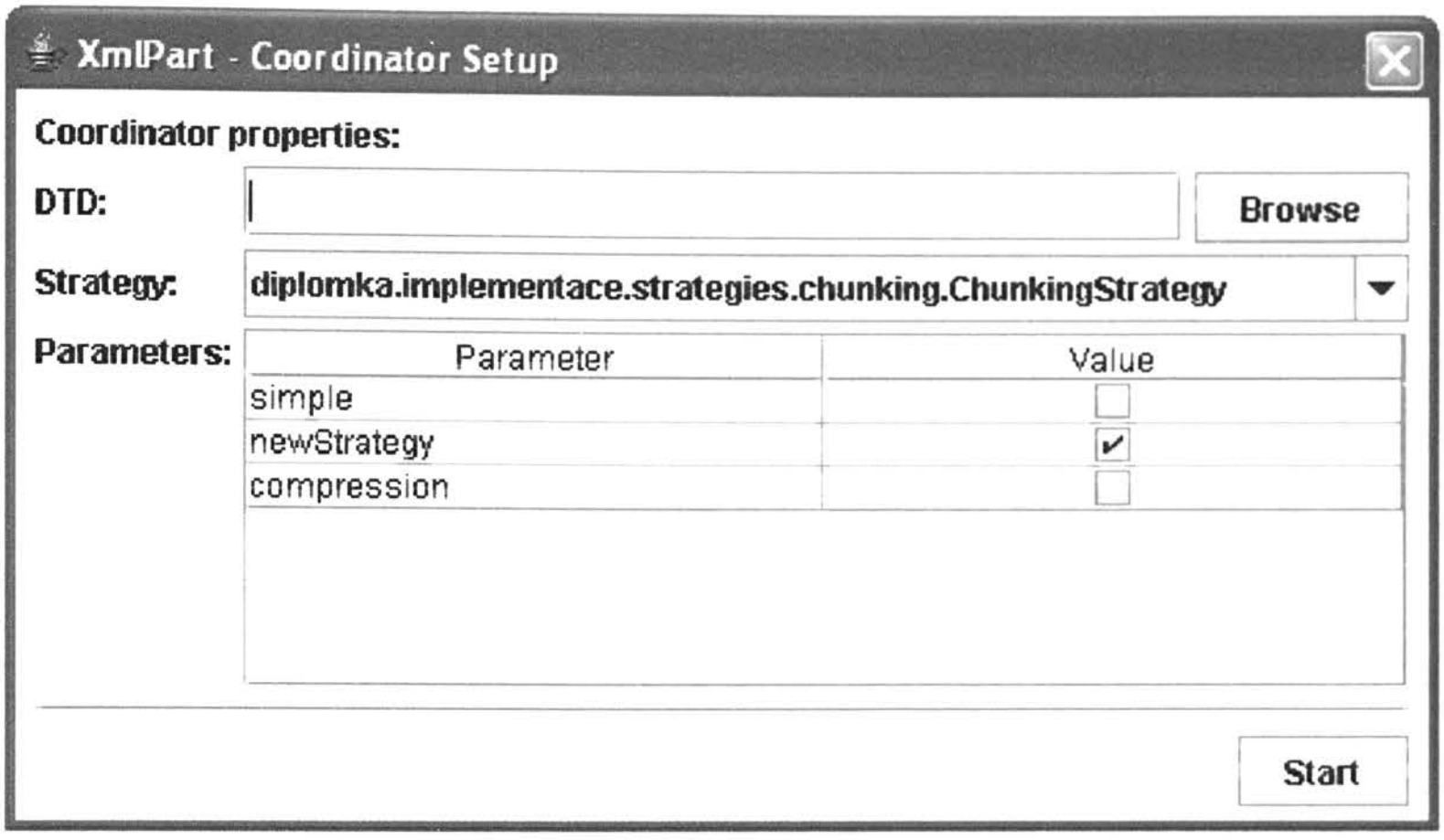
6.5.1 GUI rozhraní

GUI aplikace coordinator

Chceme-li spustit celý doručovací systém, musíme nejprve zapnout koordinátora fungujícího jako registrační služba. Přepneme se do adresáře `C:\XmlPart\program` a spustíme aplikaci z příkazové řádky příkazem

```
java -jar gui-coordinator.jar
```

Objeví se následující okno:



Obr. 13

V horní části okna zadáme cestu k DTD dokumentům, se kterými budeme pracovat. V políčku `Strategy` vybereme požadovanou strategii (identifikovanou jménem Java třídy, která ji realizuje, viz dále). Následuje výběr příslušných parametrů. Nyní shrneme jednotlivé strategie a význam jejich parametrů:

1. `diplomka.implementace.strategies.simple.SimpleStrategy`

Jednoduchá strategie, která veškeré filtrování provádí na jednom brokeru, ostatní brokery jsou nepoužité.

Parametry:

- `compression` - pokud je hodnota parametru `true`, odchozí data se na každém z brokerů komprimují

2.`diplomka.implementace.strategies.simple.SimpleDistributedStrategy`

Jednoduchá distribuovaná strategie. Dotazy se rozdělí do n skupin (n je počet brokerů - 1), každou skupinu řeší jeden broker. Zbývající broker je kořenový broker – příchozí zprávu rozešle ostatním brokerům, které se postarají o filtrování.

Parametry:

- `compression` - pokud je hodnota parametru `true`, odchozí data se na každém z brokerů komprimují

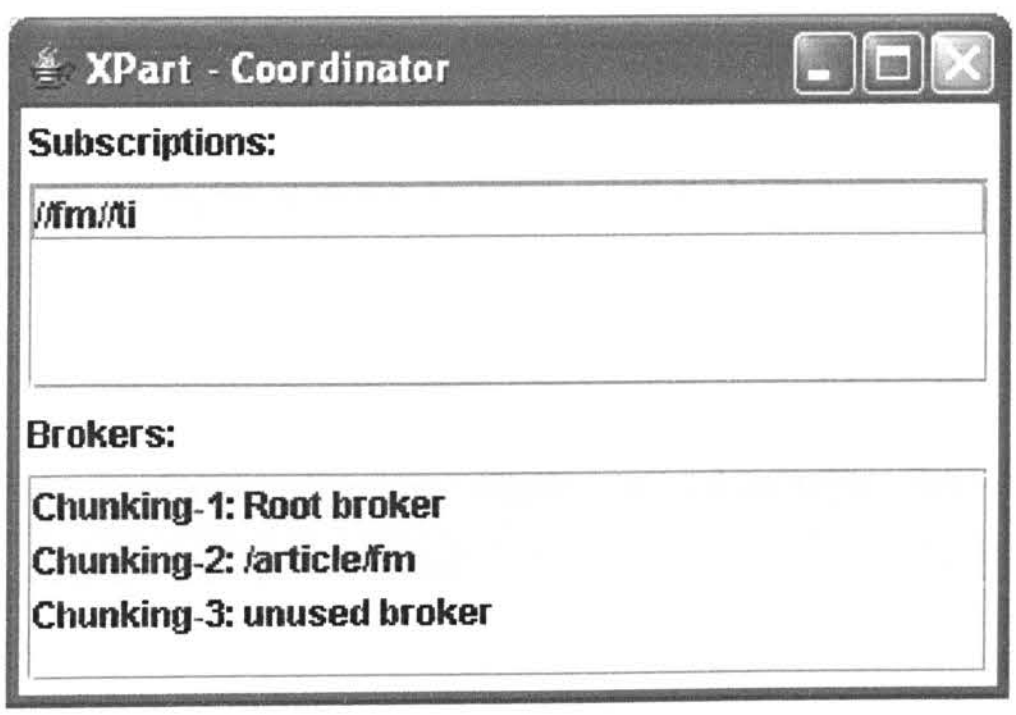
3. `diplomka.implementace.strategies.chunking.ChunkingStrategy`

Strategie využívající rozpadové schéma. Dotazy se rozdělí na skupiny, kořenový broker rozesílá rozdělená data jednotlivým brokerům. Každý z těchto brokerů pak již jen provede filtrování ve své skupině.

Parametry:

- `newStrategy` - pokud je hodnota parametru `true`, rozpadové schéma se tvoří postupem popsaným v kapitole 6.1, pokud je `false`, tvoří se rozpadové schéma "postaru" postupem popsaným v kapitole 5.3.4
- `simple` - parametr se týká přístupu k řešení vnořených cest. Pokud je hodnota parametru `true`, používá se ve strategii jednodušší z obou technik popsaných v kapitole 6.1.3, pokud je hodnota parametru `false`, používá se složitější z technik
- `compression` - pokud je hodnota parametru `true`, odchozí data se na každém z brokerů komprimují

Po zadání všech požadovaných vlastností koordinátora ho můžeme spustit stisknutím tlačítka `Start`. Objeví se následující okno, v něm můžeme jednoduchým způsobem sledovat, jak strategie přiřazuje brokery jednotlivým dotazům.



obr. 14

V horní části okna budou (po registraci uživatelů a jejich požadavků) vypsány jednotlivé dotazy, v dolní části budou vypsány brokery, poté, co se registrují v systému. Pokud klepneme na některý z dotazů, v dolní části se nám vyznačí všechny brokery, které daný dotaz zpracovávají.

Koordinátora lze kdykoliv vypnout poklepáním na křížek v pravém horním rohu okna. Vypnutí koordinátora vynutí i vypnutí všech ostatních komponent systému.

GUI aplikace broker

Po zapnutí koordinátora můžeme začít připojovat do sítě jednotlivé brokery. Spustíme ho z příkazové řádky příkazem

```
java -jar broker.jar
```

přičemž aktuální adresář musí být `C:\XmlPart\program`.

Objeví se dialogové okno, do kterého zadáme název či IP adresu počítače, na kterém běží koordinátor. Pokud koordinátor běží na témže počítači jako broker, použije se `localhost`.

Potvrdíme stiskem tlačítka `OK`, čímž se broker spustí. Objeví okno, ve kterém lze po celou dobu, kdy je broker spuštěn, sledovat shrnutí aktivit tohoto brokeru. Konkrétně jde o počet požadavků pro tento broker, celkový objem dat vstupujících do brokeru, objem dat vystupujících z brokeru a čas, který broker strávil zpracováváním jednotlivých požadavků. Stisknutím tlačítka `Refresh` lze údaje aktualizovat a stiskem tlačítka `Reset` naopak vynulovat. Broker lze kdykoli vypojit z naší sítě stiskem tlačítka `Shutdown`.

GUI aplikace subscriber

Pro spuštění příjemce zpráv se přepneme do adresáře `C:\XmlPart\program` a z příkazové řádky zadáme

```
java -jar gui-subscriber.jar
```

Objeví se totéž úvodní dialogové okno jako u brokeru; zadáme název či IP adresu počítače, na kterém běží koordinátor (podobně jako v případě brokeru, pokud koordinátor běží na témže počítači jako příjemce, použije se `localhost`), a potvrdíme stiskem tlačítka OK.

Po dobu aktivity tohoto příjemce bude zobrazeno následující okno:

obr. 15

Okno se skládá ze tří částí:

- Horní část slouží k zadání dotazu (zadá se do vstupního pole a potvrdí se stiskem ENTER).
- Střední část slouží jako přehled již zadaných dotazů. Výběrem několika dotazů a stisknutím tlačítka `Remove` se vybrané dotazy odstraní, po stisku tlačítka `Load` a výběru souboru z disku je možno načíst textový soubor s dotazy (jedna řádka odpovídá jednomu dotazu).
- Dolní část zobrazuje přijaté zprávy – vypisuje se datum přijetí a dotaz, kterému zpráva odpovídá, po stisknutí tlačítka `View` (nebo stačí jen ENTER) se zobrazí obsah zprávy, stiskem tlačítka `Delete` je pak možné odstranit staré zprávy.

Celá aplikace se ukončí poklepáním na křížek v pravém horním rohu.

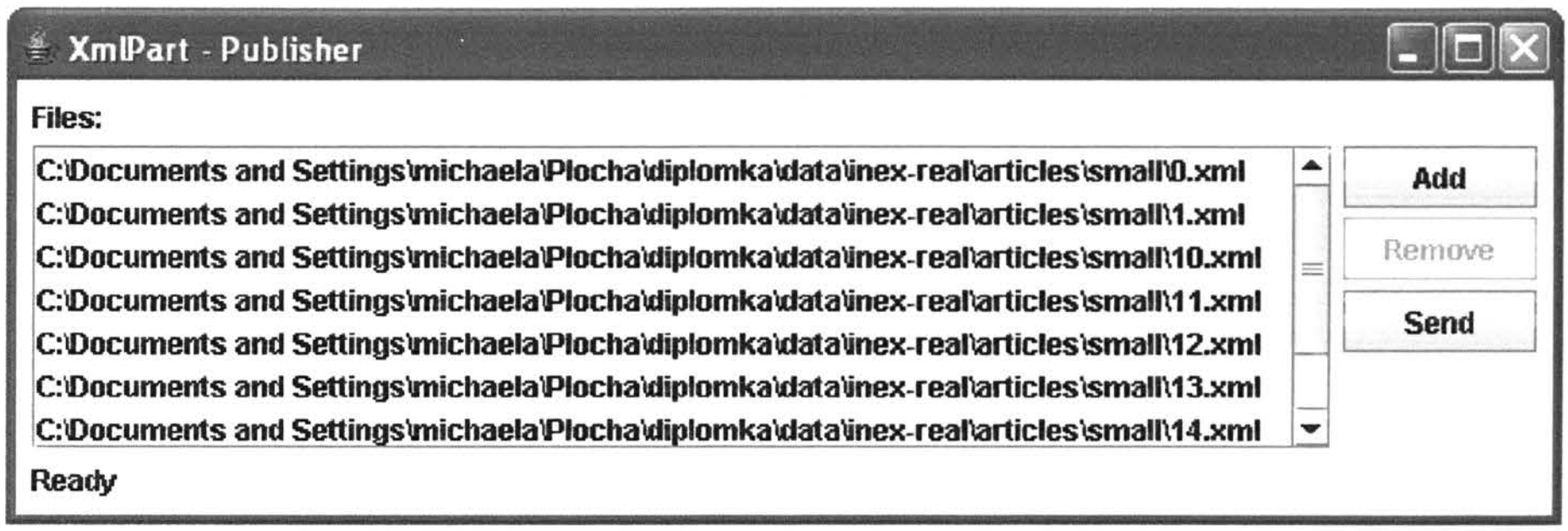
GUI aplikace publisher

Pro spuštění poskytovatele zpráv se přepneme do adresáře `C:\XmlPart\program` a z příkazové řádky zadáme

```
java -jar gui-subscriber.jar
```

Opět se objeví úvodní dialogové okno; zadáme název či IP adresu počítače, na kterém běží koordinátor (stejně jako v případě minulých dvou aplikací, pokud koordinátor běží na témže počítači jako poskytovatel, použije se `localhost`) a potvrdíme stiskem tlačítka `OK`.

Po spuštění poskytovatele můžeme jeho činnost sledovat řídit v oknu podobném následujícímu:



Obr. 16

Stisknutím tlačítka `Add` lze přidat dokumenty do okna (podržením klávesy Shift jich lze vybrat libovolné množství ve zvoleném adresáři), pro poslání těchto dokumentů do systému stiskneme tlačítko `Send`. Před příštím odesláním dat od tohoto poskytovatele lze vybranou dokumentovou sadu upravit - stiskem tlačítka `Remove` dokument odstraníme ze sady připravené k odeslání (přičemž můžeme podržením klávesy Shift označit k odstranění i více dokumentů), stiskem tlačítka `Add` lze dokumenty přidávat.

Aplikaci můžeme ukončit poklepáním na křížek v pravém horním rohu.

6.5.2 Dávkové programy

Pro účely měření výkonnosti systému jsme naimplementovali také dávkové varianty jednotlivých aplikací.

Aplikace coordinator

Pro spuštění koordinátora se přepneme do adresáře `C:\XmlPart\program` a z příkazové řádky zadáme

`java -jar coordinator.jar <dtd> <strategie> <parametry>`,

kde `<dtd>` je cesta k dtd dokumentům, se kterými budeme pracovat, strategie je použitá strategie (název java třídy, viz podkapitola 6.5.1 v sekci gui-coordinator) a `<parametry>`

označují mezerami oddělené parametry strategie ve formě `parametr=hodnota` (opět viz kapitola 6.5.1) Po spuštění aplikace se po době nutné k tomu, aby parser zanalyzoval zadané DTD, objeví text

```
C:\XmlPart\program>java -jar coordinator.jar  ..\testovaci-
data\articles\xmlarticle.dtd
diplomka.implementace.strategies.chunking.ChunkingStrategy
newStrategy=true simple=false compression=false

Koordinator bezi. Zmacknete q<ENTER> pro ukonceni.
```

symbolizující, že koordinátor je nastartován a je tedy možné spustit ostatní komponenty systému. Koordinátora je možno ukončit stiskem klávesy q a ENTER, čímž se zároveň automaticky zajistí ukončení ostatních aplikací.

Aplikace broker

Pro spuštění brokeru se přepneme do adresáře `C:\XmlPart\program` a z příkazové řádky zadáme

`java -jar broker.jar <koordinátor>`,

kde `<koordinátor>` je jméno nebo IP adresa počítače, na kterém běží koordinátor. Pokud koordinátor běží na témže počítači jako broker, použije se `localhost`.

```
C:\XmlPart\program>java -jar broker.jar localhost

Broker bezi. Zmacknete q<ENTER> pro ukonceni.
```

Činnost brokera ukončíme podobně jako v předchozím případě stiskem klávesy q a ENTER.

Aplikace subscriber

Pro spuštění příjemce se přepneme do adresáře `C:\XmlPart\program` a z příkazové řádky zadáme

`java -jar subscriber.jar <koordinátor> <soubor_s_dotazy>`

kde `<koordinátor>` je jméno nebo IP adresa počítače, na kterém běží koordinátor (pokud koordinátor běží na témže počítači jako poskytovatel, použije se `localhost`), a `<soubor_s_dotazy>` je cesta k textovému souboru s dotazy (jedna řádka odpovídá jednomu dotazu).

```
C:\XmlPart\program>java -jar subscriber.jar localhost
..\testovaci-data\dotazy.txt

Subscriber bezi. Zmacknete q<ENTER> pro ukonceni.
```

Stisk klávesy q následovaný stiskem klávesy ENTER aplikaci ukončí

Aplikace publisher

Pro spuštění příjemce se přepneme do adresáře `C:\XmlPart\program` a z příkazové řádky zadáme

`java -jar publisher.jar <koordinátor> <složka>`

kde `<koordinátor>` je jméno nebo IP adresa počítače, na kterém běží koordinátor (pokud koordinátor běží na témže počítači jako poskytovatel, použije se `localhost`), a `<složka>` je cesta k adresáři s XML dokumenty.

```
C:\XmlPart\program>java -jar publisher.jar localhost
..\testovaci-data\small

Publisher bezi. Zmacknete q<ENTER> pro ukonceni, s<ENTER> pro
poslani dat.
```

Po stisku klávesy s poskytovatel odešle všechny soubory v zadané složce doručovací službě:

```
          posilam ..\data\X1037.XML
          posilam ..\data\X1027.XML
          posilam ..\data\X1020.XML
          posilam ..\data\X1047.XML
          posilam ..\data\X1055.XML
soubory odeslany.
Publisher bezi. Zmacknete q<ENTER> pro ukonceni, s<ENTER> pro
poslani dat.
```

Dodejme jen, že stisk `q` následovaný stiskem klávesy `ENTER` aplikaci ukončí.

6.6 Technické řešení

6.6.1 Architektura systému

Jak již bylo řečeno, celý systém se skládá ze čtyř aplikací napsaných v programovacím jazyku Java. Jednotlivé aplikace spolu komunikují pomocí vrstvy Java RMI, která umožňuje publikovat objekty tak, že jejich metody je pak možné zavolat zvenčí. Jednotlivé aplikace tak tedy odpovídají implementacím čtyř základních rozhraní systému:

- `ICoordinator` – koordinátor. Obsahuje metody pro registraci a odregistrování brokerů, poskytovatelů a příjemců. Jeho součástí je routovací strategie (`IRoutingStrategy`), která se stará o nastavování konfigurace routovacích tabulek (`IRoutingTableConfig`) na jednotlivých brokerech.
- `IBroker` – broker. Obsahuje metodu `receiveMessage`, kterou přijme zaslanou zprávu. Tuto zprávu pak předloží routovací tabulce (`IRoutingTable`), která na základě obsahu zprávy rozešle její kopie nebo jejich části dalším v řadě – může se jednat o další brokery nebo příjemce zpráv.
- `ISubscriber` – příjemce zpráv. Jeho jedinou metodou je metoda `receiveMessage()`, pomocí které přijímá zprávy.
- `IPublisher` – poskytovatel zpráv. Poté, co se úspěšně registruje u koordinátora, je mu prostřednictvím metody `setMessgeReceiver()` nastaven broker, kterému bude posílat zprávy. Pokud v systému žádný broker není, bude nastaven null.

Mezi další důležitá rozhraní patří:

- `IXmlMessage` – zpráva, kterou posílá producent nebo broker příjemci zpráv nebo jinému brokeru. Existuje několik implementací v závislosti na použitých technikách (formát souboru, komprese).
- `IXmlWriter` – slouží k zapisování XML dokumentů jednotnou formou bez ohledu na konkrétní implementaci `IXmlMessage`.

6.6.2 Komponenty systému

Protože programujeme v objektovém prostředí, bylo vhodné rozdělit systém do několika komponent:

- implementace jednotlivých aplikací (GUI a dávkové)
- routovací strategie
- implementace routovacích tabulek
- XPathParser

6.6.3 Použité technologie

- Programovací jazyk a runtime: Java 5.0
- Knihovny: yfilter.jar (filtrovací systém YFilter), java_cup.jar (pomocná knihovna YFilteru), dtdparser113.jar (DTD parser, pomocná knihovna YFilteru)
- Vývojové prostředí: Eclipse 3.1
- Nástroj pro tvorbu balíků a sestavení aplikace: Ant 1.6
- Generování parserů pro XPath ze zadané gramatiky: JFlex (tokenizer regulárního jazyka) a BYACC/J (parser bezkontextové gramatiky)

6.7 Experimentální výsledky

Cílem praktických experimentů se systémem XmlPart bylo porovnání implementovaných strategií vzhledem k času potřebnému na filtrování dokumentu, množství dat vyměněných mezi jednotlivými komponentami systému a počtu použitých brokerů.

Konkrétně porovnáváme následující strategie:

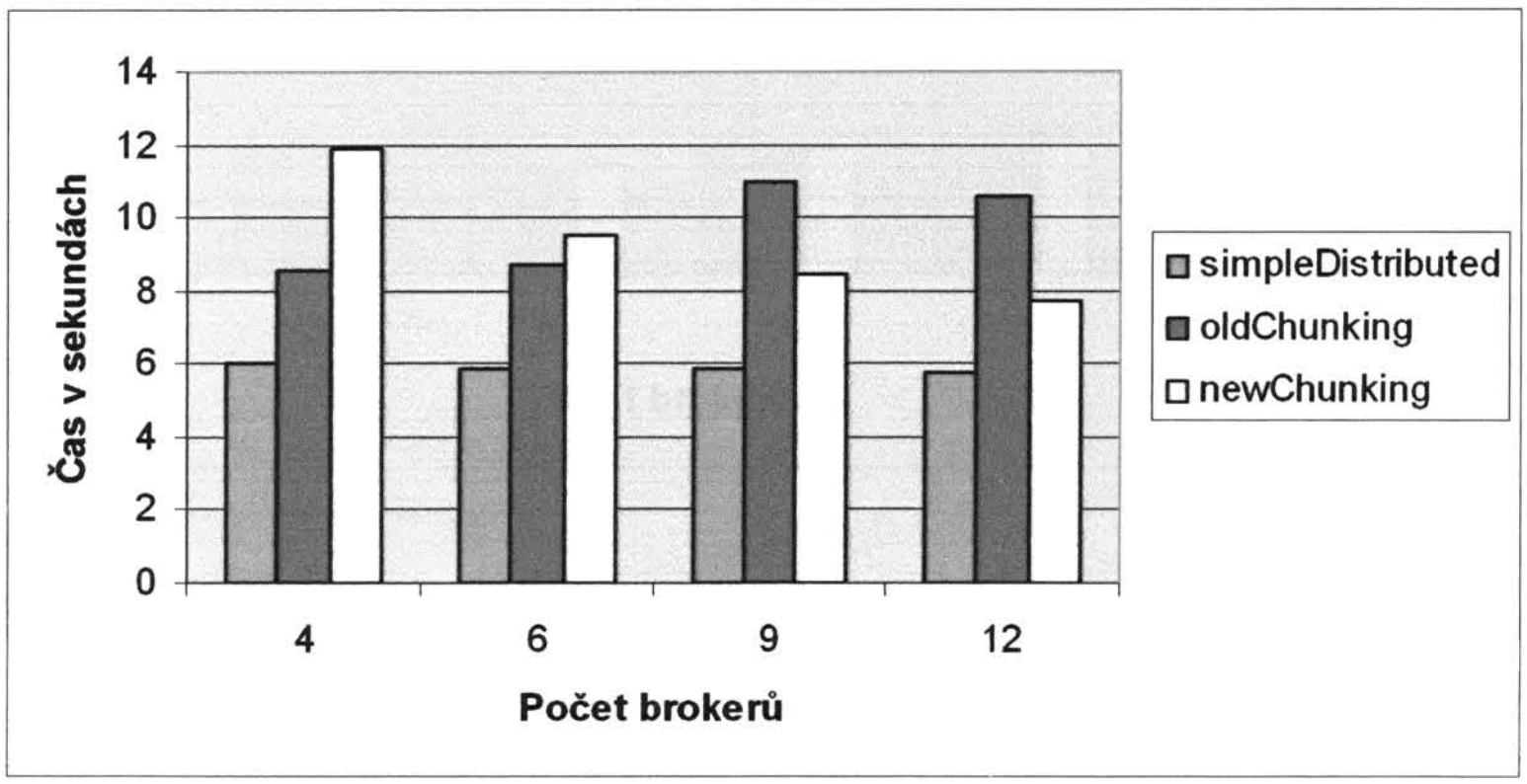
- jednoduchou strategii, kdy jeden kořenový broker rozesílá veškerá data všem připojeným hostitelským brokerům, na kterých probíhá samotná filtrace (v grafech je tato strategie označena jako `simpleDistributed`)
- strategii popsanou v kapitole 5.3.4 (v grafech označena jako `oldChunking`)
- námi navrženou strategii popsanou v kapitole 6.1 (v grafech označena jako `newChunking`)

Věnujeme se rovněž vlivu komprese na výsledky jednotlivých měření a způsobu řešení vnořených cest u strategií využívajících rozpadové schéma.

Experimenty se systémem XmlPart byly prováděny na počítačích s procesorem Pentium 4, 2.6 GHz, 512 MB RAM, pod operačním systémem Linux. Pro měření jsme si vygenerovali 2000 dotazů. Použili jsme 20 dokumentů, které jsme získali úpravou dokumentů z databáze INEX tím, že jsme původní dokumenty zřetězili a přidali nový kořenový element tak, aby výsledné dokumenty měly velikost přibližně 500 kB - 1000 kB. Použité dokumenty jsou i s příslušným DTD uloženy na doprovodném CD (viz Dodatek B).

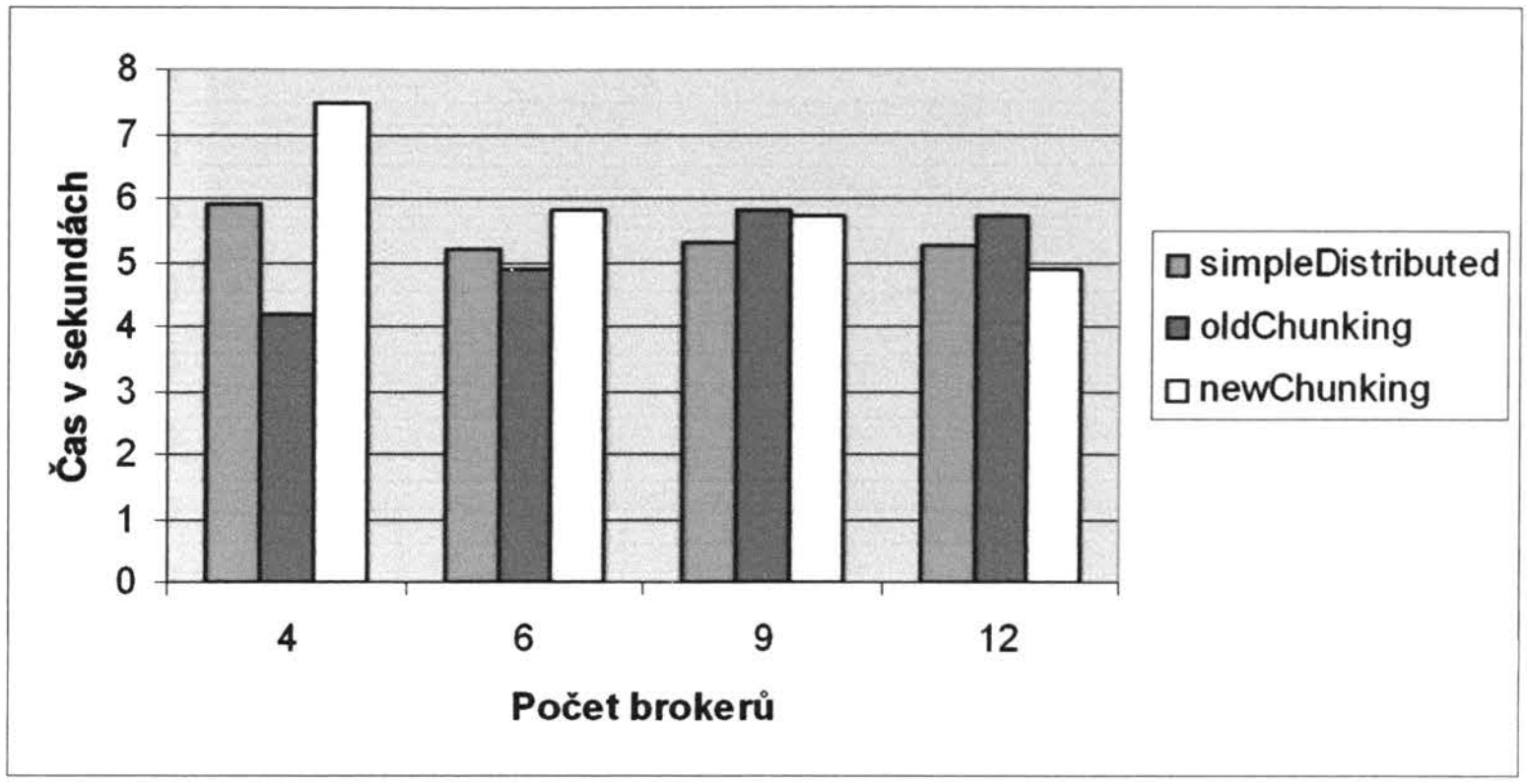
6.7.1 Porovnání z hlediska času

Nejprve se věnujme vlivu počtu použitých brokerů a jednotlivých strategií na čas zpracování. Čas jsme analyzovali z několika hledisek - nejprve šlo o maximální čas zpracování (graf 1). Tento určujeme tak, že k času, který spotřebuje ke své práci kořenový broker, přičteme maximum z časů spotřebovaných jednotlivými hostitelskými brokery ke zpracování příslušejících skupin dotazů.



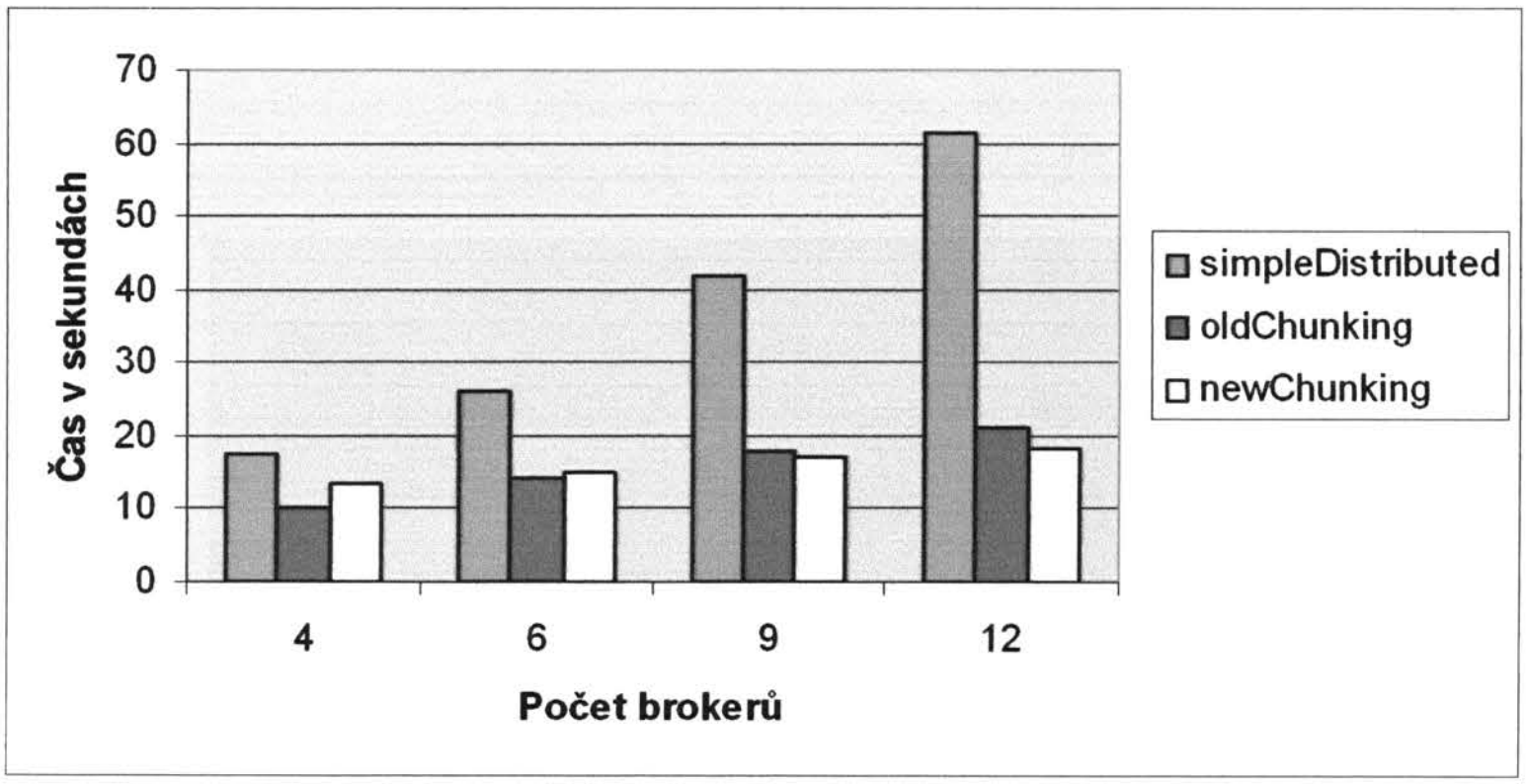
Graf 1

Dotazů zpracovávaných maximální dobu bývá poměrně málo, zajímavé je tedy zkoumat průměrnou dobu zpracování jednoho dotazu. Hodnoty jsou vyneseny v grafu 2.



Graf 2

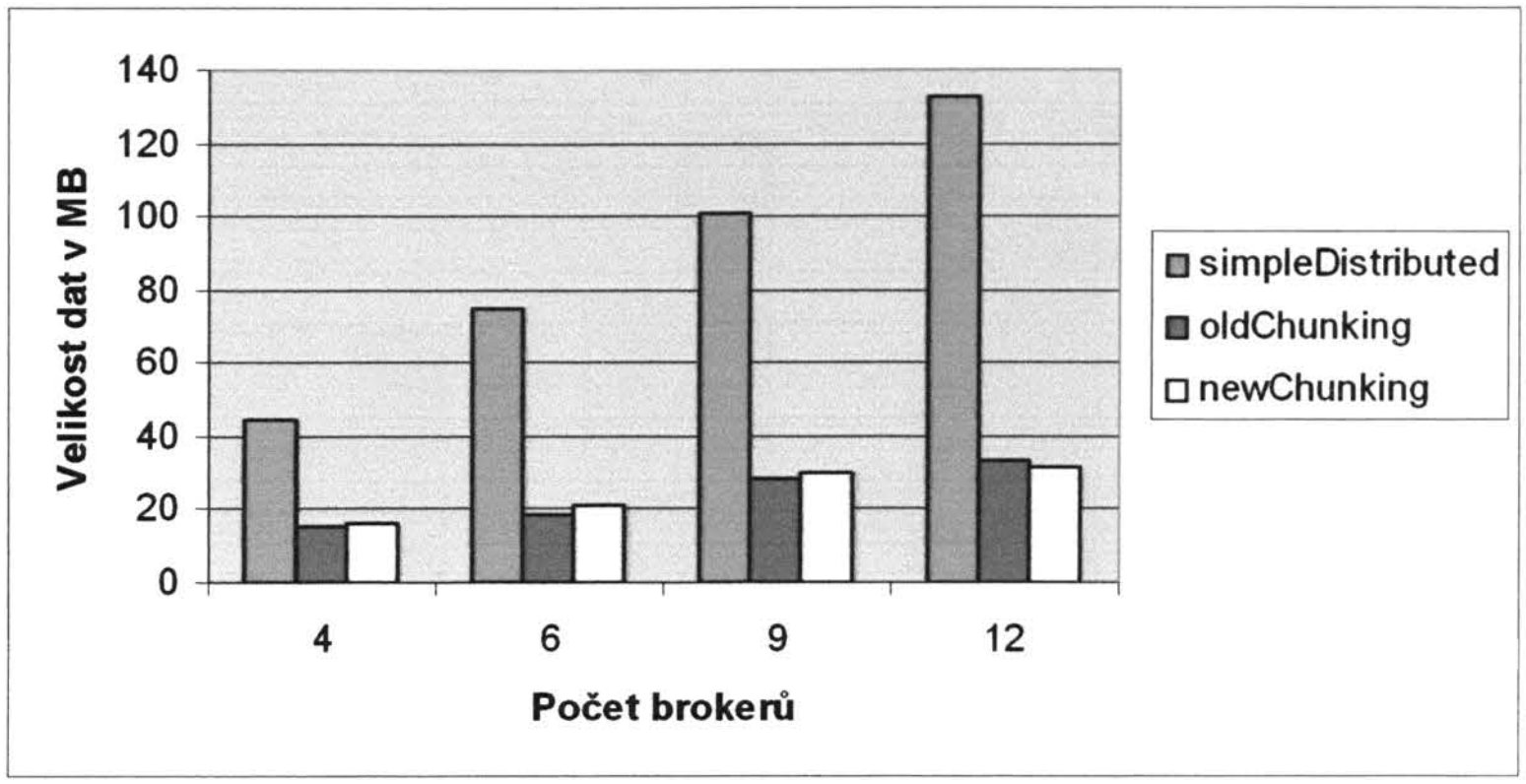
Další srovnání (graf 3) se týká celkového času spotřebovaného všemi brokery - podle tohoto údaje lze posuzovat celkovou zátěž systému při zpracování požadavku.



Graf 3

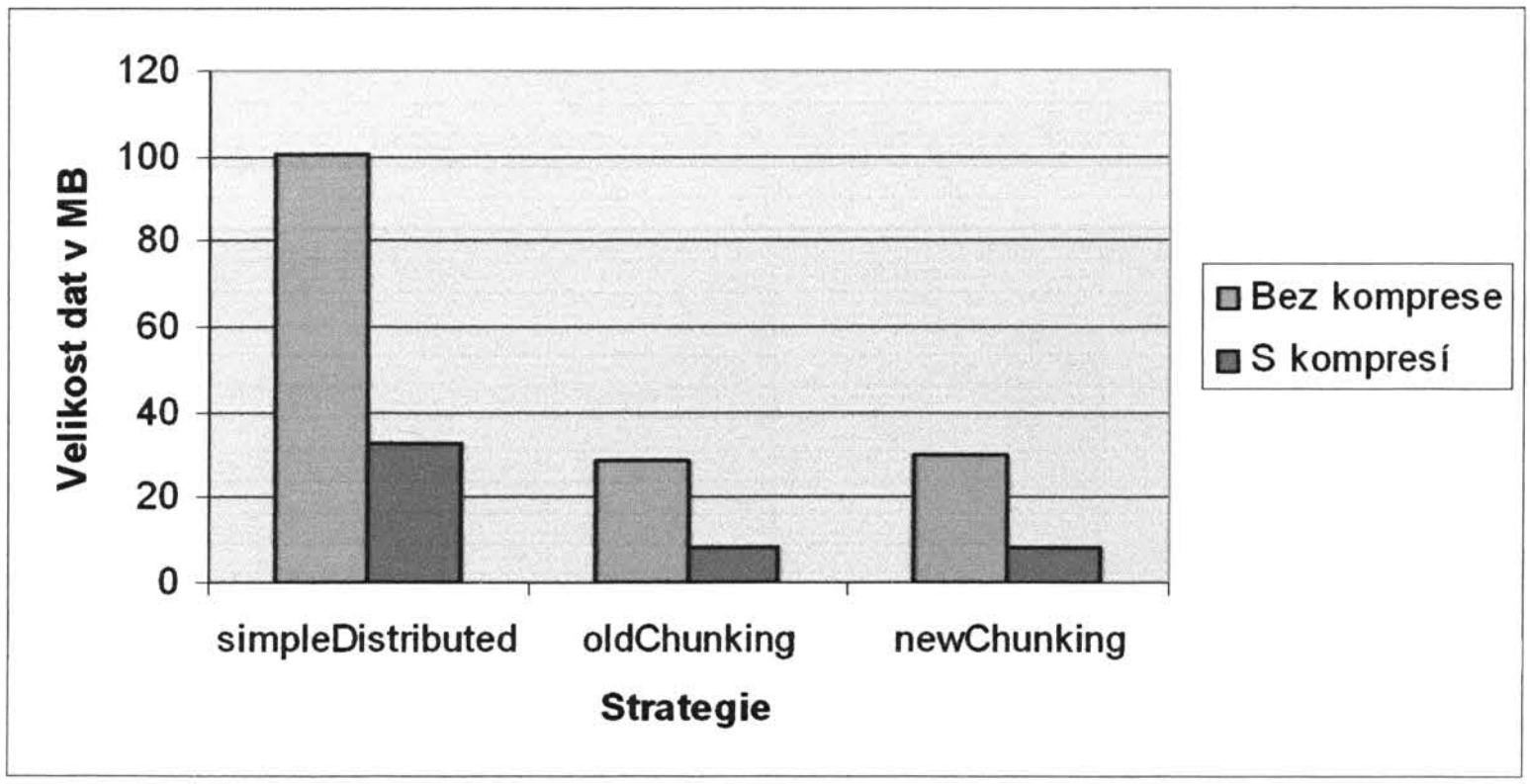
6.7.2 Porovnání z hlediska přenesených dat

Neméně důležitým kritériem výkonnosti jednotlivých strategií je celkový objem dat vyměněných mezi jednotlivými brokery (graf 4).



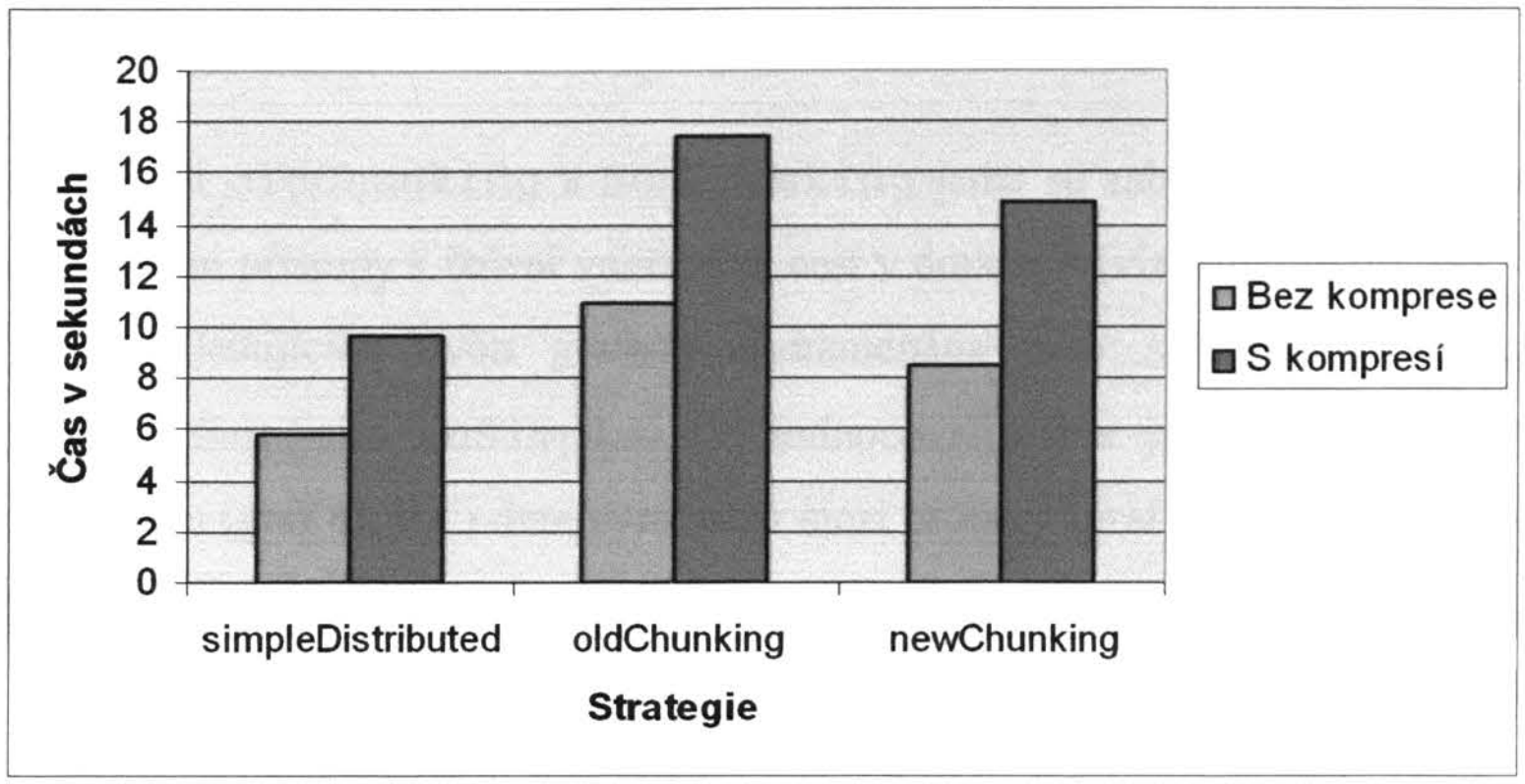
Graf 4

Z minulého grafu je patrné, že zejména při použití strategie označené `simpleDist` dochází mezi jednotlivými brokery k výměně skutečně velkého objemu dat. Zkoumali jsme tedy, jak moc by tento problém pomohla vyřešit komprese (graf 5), pro tato měření bylo použito 9 brokerů.



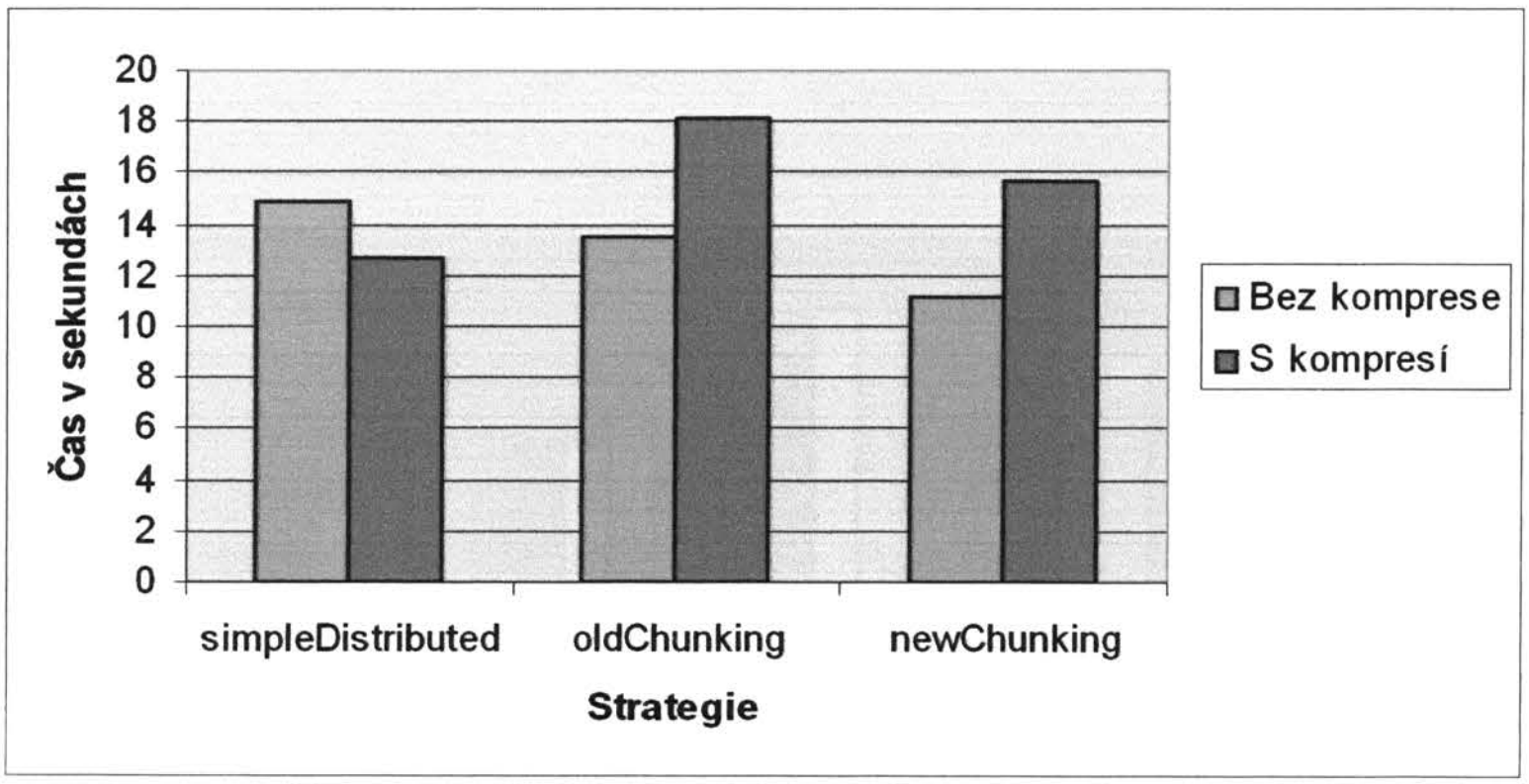
Graf 5

Výsledky z minulého grafu vypadají velice povzbudivě, komprese výrazně snížila objem vyměněných dat. Komprese má ale i svou stinnou stránku - čas ke zpracování dotazu se navyšuje o čas potřebný ke kompresi (graf 6).



Graf 6

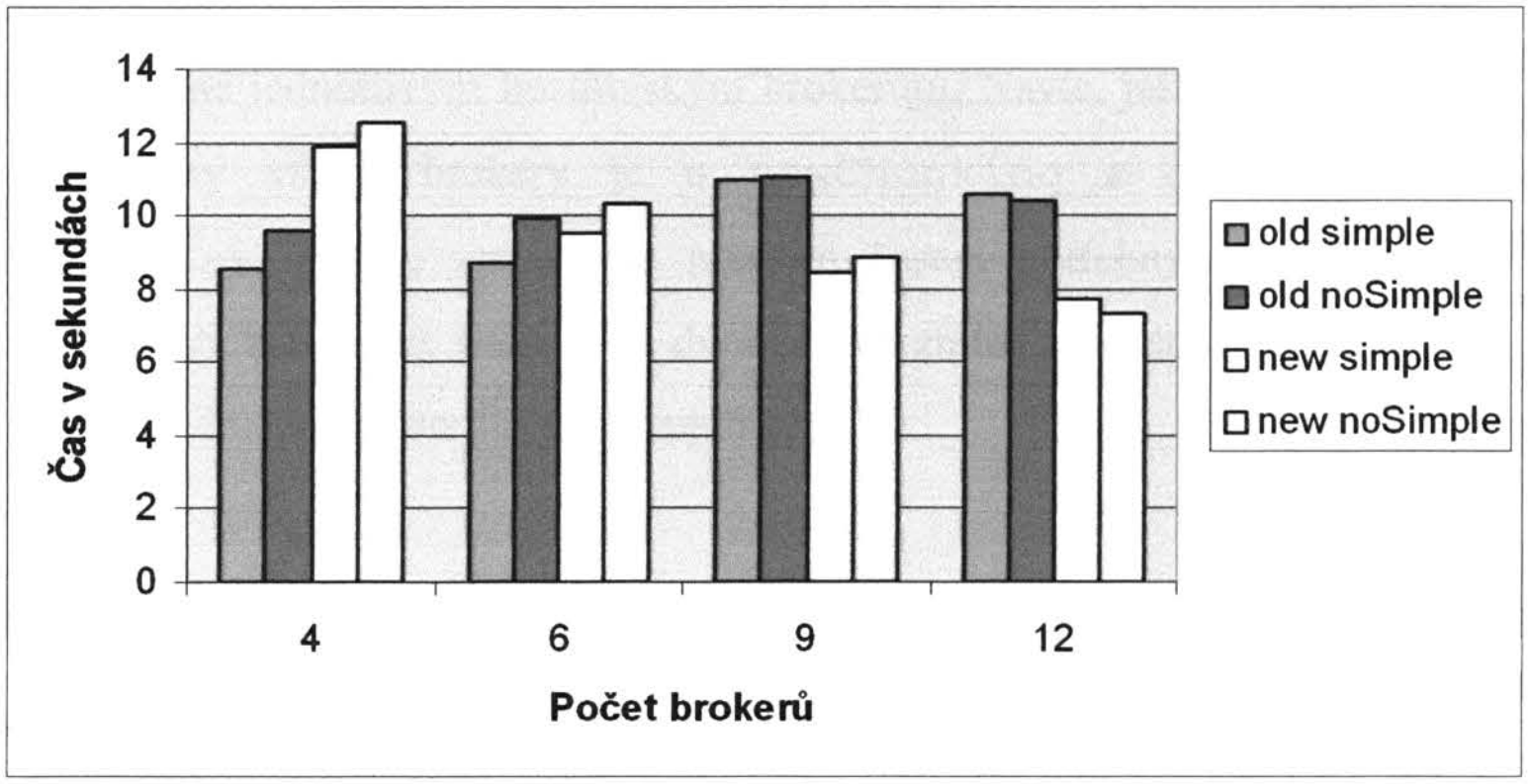
Známe-li přenosovou rychlost sítě, můžeme dvě posledně zmíněná hlediska použít pro odhad skutečného času potřebného ke zpracování požadavku (skutečným časem míníme, že zohledňujeme jak čas spotřebovaný prací jednotlivých brokerů, tak i čas potřebný k výměně dat mezi jednotlivými brokery). Z následujícího grafu 7 můžeme posuzovat, zda se komprese u jednotlivých strategií z hlediska tohoto času skutečně vyplatí. Testovali jsme na síti s přenosovou rychlostí 11,2 MB/s.



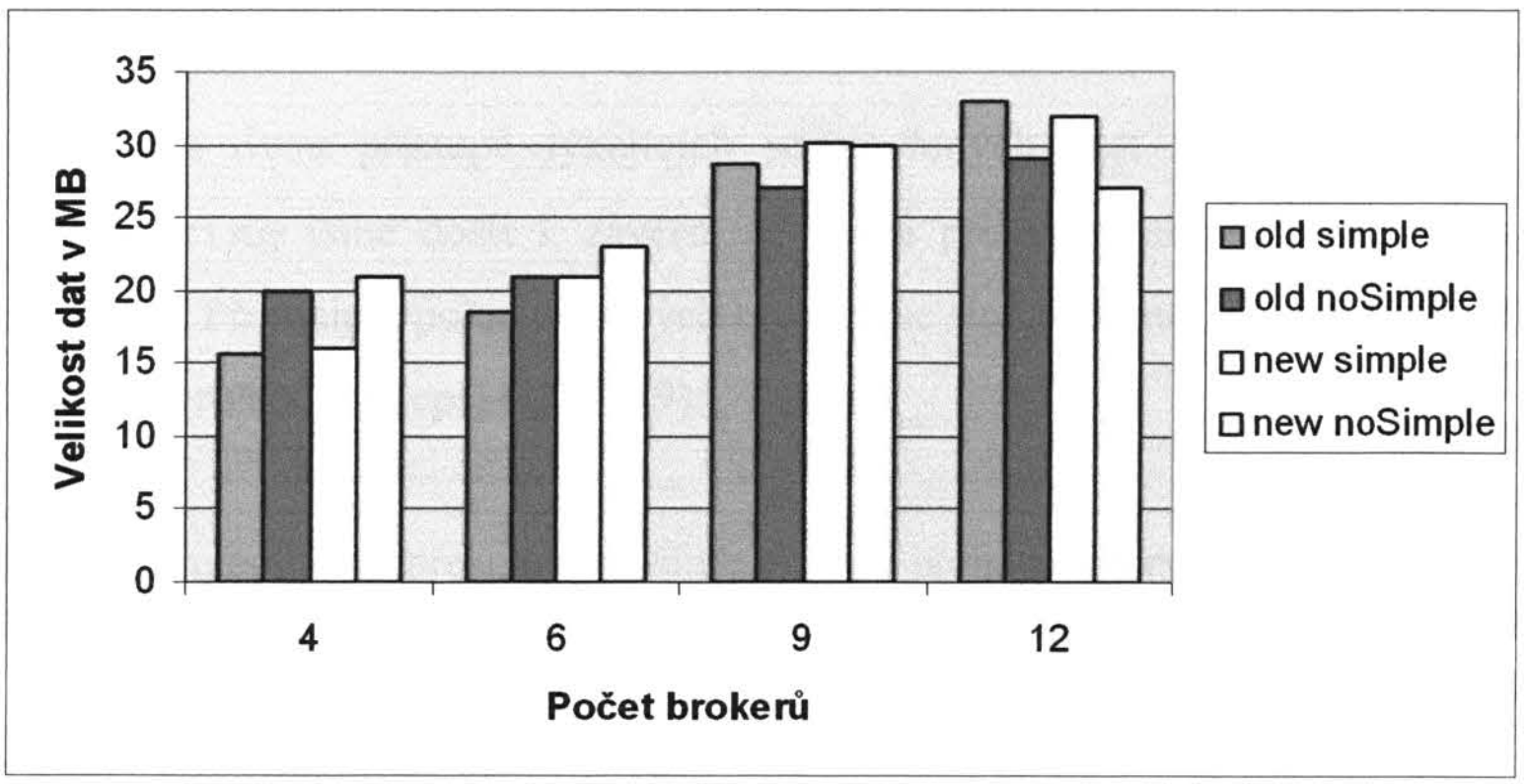
Graf 7

6.7.3 Porovnání z hlediska přístupu k vnořeným cestám

U strategií `oldChunking` a `newChunking` jsme se zabývali také rozdílem mezi oběma navrženými přístupy k řešení vnořených cest v dotazech (viz kapitola 6.1.3) - jednodušší (tato je v následujících dvou grafech zaznamenána jako `simple`) i složitější technikou (zaznamenána jako `noSimple`). Vyhodnocovali jsme jak čas potřebný ke zpracování požadavku (graf 8), tak i data vyměněná mezi brokery (graf 9). Při měření jsme zohledňovali počet použitých brokerů.



Graf 8



Graf 9

6.7.4 Vyhodnocení experimentů

Nyní již můžeme shrnout výsledky provedených experimentů. Strategie `simpleDist` vychází proti oběma ostatním strategiím lépe co se týče času potřebného ke zpracování požadavku (graf 1), ale naopak vychází ze srovnání zdaleka nejhůře z hlediska celkového objemu přenesených dat mezi brokery (graf 4). Krátká doba je umožněna tím, že v kořenovém brokeru nedochází k žádnému zdržení - veškerá data jsou bez jakéhokoli zpracování a filtrace odeslána dále na hostitelské brokery, což ale na druhou stranu zapřičiňuje zmíněný vysoký objem vyměněných dat. Naopak u strategií `newChunking` a `oldChunking` zajišťují menší objem dat za cenu vyššího celkového času způsobeného rozdělením původního dokumentu na části posílané jednotlivým hostitelským brokerům. Navíc, jak prokazuje graf 3, celkový čas spotřebovaný všemi brokery je u `newChunking` a `oldChunking` nižší než u `simpleDist`, což je způsobeno menším časem potřebným k samotné filtraci dat na hostitelských brokerech (což je u dvou dříve zmíněných strategií způsobeno dostatečnou podobností dotazu v jednotlivých skupinách).

Strategie `newChunking` se oproti strategii `oldChunking` vyplatí v situaci, kdy máme k dispozici více brokerů, a to jak z hlediska přenesených dat (graf 4), tak i z hlediska času (tendence je patrna z grafů 1, 2 i 3). Podrobnější analýzou výsledků jsme usoudili, že toto je způsobeno zejména menší velikostí směsné skupiny u strategie `newChunking`, hostitelský broker s touto směsnou skupinou tedy spotřebuje na její zpracování menší čas.

Porovnáním dvou přístupů týkajících se vnořených cest u strategií `oldChunking` a `newChunking` jsme došli k závěru, že volba přístupu nehraje v celkovém efektu příliš velkou roli. Při malém počtu použitých brokerů se složitější metoda zdá být horší a při vyšším počtu brokerů naopak lepší (graf 8, 9).

Pokud zvažujeme kompresi, tak musíme uvážit i přenosovou rychlost sítě, na které pracujeme. Z grafu 7 je patrno, že pro naši konkrétní síť s přenosovou rychlostí 11,2 MB/s se komprese vyplatí pro strategii `simpleDist`, ale už nikoli pro strategie `oldChunking` a `newChunking` - úspora času potřebného k přenosu komprimovaných dat byla nižší než čas potřebný k vlastní kompresi.

7 Závěr

Tato práce se zabývala problematikou publish-subscribe systémů - systémů zajišťujících filtrování informací a jejich doručování jednotlivým zájemcům. Po základním seznámení s hlavními myšlenkami publish-subscribe systémů jsme se dále zaměřili na systémy pracující se zprávami zapsanými v jazyku XML. Proto jsme museli prostudovat základní vlastnosti XML i možnosti některých důležitých nástrojů pro efektivní práci s daty v tomto formátu.

Seznámili jsme se s technikami vlastního filtrování XML dokumentů, kdy sadu dotazů zapsaných v jazyku XPath testujeme na shodu s daným XML dokumentem. Zmínili jsme některé reálné filtrovací systémy, přičemž s hlavními myšlenkami momentálně pravděpodobně nejznámějšího XML filtrovače - YFilteru - jsme se seznámili důkladněji. Následně jsme probrali základní služby publish-subscribe systémů, které poskytují určitou nadstavbu nad samotným filtrovačem XML tím, že řeší i problematiku registrování jednotlivých žadatelů o informace a efektivní doručování těchto informací v reálných sítích.

Navrhli jsme a v prototypové verzi naimplementovali jednoduchý publish-subscribe systém XmlPart. Pro něj jsme navrhli několik různých strategií, zejména jsme se věnovali návrhu strategie využívající nový způsob tvorby rozpadového schématu. Následně jsme strategie implementovali, vyzkoušeli na reálných datech a porovnali vzhledem k různým kritériím (zejména jsme si všímali času potřebného na filtrování dokumentu a množství dat vyměněných mezi jednotlivými komponentami systému). Experimenty prokázaly klady i zápory každé z navržených strategií.

Máme zato, že oba předem vytyčené cíle - zmapování dané oblasti i implementace vlastního publish-subscribe systému - se podařilo úspěšně naplnit.

Použitá literatura

[1] Altinel, M., Franklin, M.: Efficient Filtering of XML Documents for Selective Dissemination of Information. VLDB, 2000.

[2] Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R., Sturman, D.: An efficient multicast protocol for content-based publish-subscribe systems. ICDCS 1999.

[3] Bruno, N., Gravano, L., Koudas, N., Srivastava, D.: Navigation- vs. Index-Based XML Multi-Query Processing. ICDE 2003.

[4] Carzaniga, A., Rosenblum, D., Wolf, A.: Achieving scalability and expressiveness in an Internet-scale event notification service. ACM 2000.

[5] Chan, Y., Felber, P.: A Scalable Protocol for Content-Based Routing in Overlay Networks. NCA, 2003.

[6] Chan, Y., Felber, P., Garofalakis, M., Rastogi, R.: Efficient filtering of XML documents with XPath expressions. ICDE 2002

[7] Chen, J., DeWitt, D., Tian, F., Wang, Y.: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. 2000.

[8] Cugola, G., Nitto, E. D., Fugetta, A. The Jedi event-based infrastructure and its application to the development of the opss wfms. IEE Trans. Softw. Eng., 2001.

[9] Diao, Y., Franklin, M.: Query Processing for High-Volume XML Message Brokering. VLDB, 2003.

[10] Diao, Y., Rizvi, S., Franklin, M.: Towards an Internet-Scale XML Dissemination Service. VLDB, 2004.

[11] Eugster, P, Felber, P., Guerraoui, R., Kermarrec, A.: The Many Faces of Publish/Subscribe. ACM Computing Surveys, 2003.

[12] Hopcroft, J. E., and Ullman, J. D. : Introduction to Automata Theory, Languages and Computation, 1979.

[13] Kosek, J: Seriál o XML pro Softwarové noviny. URL: http://www.kosek.cz/clanky/swn-xml

[14] Kosek, J.: XML pro každého. URL: http://www.kosek.cz/xml

[15] Kosek, J. : XML schémata. URL: http://www.kosek.cz/xml/schema

[16] Kosek, J.: XSLT v příkladech. http://www.kosek.cz/xml/xslt

[17] Lehner, W., Irmert, F.: XPath-Aware Chunking of XML Documents. BTW-Konferenz, 2003.

[18] Liefke, H., Suciu, D.: XMILL: An Efficient Compressor for XML Data. SIGMOD 2000.

[19] Oki, B., Pfluegl, M., Siegel, A., Skeen, D. - The information bus - an architecture for extensible distributed systems. ACM SIGOPS, 1993.

[20] Papaemmanouil, O., Cetintemel, U.: SemCast: Semantic Multicast for Content-based Data Dissemination, 2004.

[21] Pokorný, J: Dotazovací jazyky. Skripta MFF UK, Nakladatelství Karolinum, 2002

[22] Segall, B., Arnold, D. Elvin has left the buliding: A publish/subscribe notification service with quenching. AUUG 1997.

[23] The official website for SAX. URL: http://www.saxproject.org

[24] Wood, P.: Containment for XPath fragments under DTD constraints. ICDT, 2003.

[25] World Wide Web Consortium (W3C), URL: http://www.w3.org

[26] World Wide Web Consortium (W3C): Document Object Model (DOM). URL: http://www.w3.org/DOM

[27] World Wide Web Consortium (W3C): Extensible Markup Language (XML). URL: http://www.w3.org/XML

[28] World Wide Web Consortium (W3C): XML Path Language (XPath). URL: http://www.w3.org/TR/xpath

Dodatek A: Gramatika systému XmlPart

```
[1]  PathExpr          ::=   (( "/" | "//" ) StepExpr )+ ( "/"
     AdditionalSelect )?
[2]  StepExpr          ::=   ( QName | "*") ( "[" Predicate "]" )*
[3]  Predicate         ::=   SimplePredicate | PathPredicate
[4]  SimplePredicate   ::=    "@" QName "=" ( Literal | PositiveNumber)
     | PositiveNumber
     | "position" "(" ")" Comparator PositiveNumber
     | "text" "(" ")" "=" ( Literal | PositiveNumber )
[5]  PathPredicate     ::=    ( ".//" | "./" )? SimpleStepExpr
     (( "/" | "//" ) SimpleStepExpr )*
[6]  SimpleStepExpr    ::=    ( QName | "*") ( "[" SimplePredicate "]" )*
[7]  AdditionalSelect  ::=    "@" QName | "text" "(" ")"
[8]  Comparator        ::=    "=" | "!=" | ">" | "<" | ">=" | "<="
[9]  QName             ::=    NCNAME
[10] NCNAME            ::=    ( Letter | '_' ) ( NCNameChar )*
[11] NCNameChar        ::=    Letter | Digit | '.' | '-' | '_'
[12] Literal           ::=    "" [^"]* ""
[13] PositiveNumber    ::=    [1-9][0-9]*
[14] Digit             ::=    [0-9]
[15] Letter            ::=    [a-zA-Z]
```

Dodatek B: Obsah přiloženého CD-ROM

Součástí této diplomové práce je přiložené CD s následujícím obsahem:

- `program` - JAR soubory prototypu XmlPart
- `zdrojovy-kod` - zdrojové soubory prototypu XmlPart
- `testovaci-data` - složky s testovacími XML soubory, příslušné DTD a sada dotazů
- `java-1.5` - Java 5.0, verze pro MS Windows