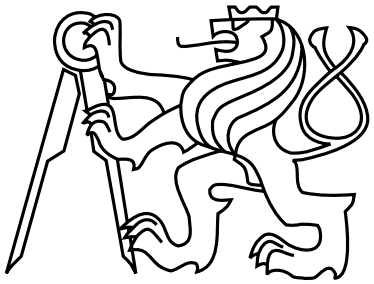
Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Zefektivnění komunikační zátěže protokolu XML

Bc. Petr Dousek

Vedoucí práce: Ing. Peter Macejko

Studijní program: Otevřená informatika, Magisterský

Obor: Počítačové inženýrství

12. května 2014

Poděkování

Děkuji vedoucímu diplomové práce Ing. Peteru Macejkovi za vedení práce a za poskytnutí cenných rad a připomínek nutných k vypracování této diplomové práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 12. 5. 2014 .....

Abstract

The aim of this Master's thesis is a research into possibilities for optimization of XML protocol's network and computational complexity, a benchmark testing of previously described methods for XML protocol optimization and an integration of those methods into existing technologies/frameworks which rely on XML protocol. Fast Infoset and Efficient XML Interchange Recommendations/Standards which are aimed at efficient communication via XML protocol, are described in the first part of the thesis. The second part of the thesis contains results of benchmark testing of previously mentioned Standards from the size reduction and computation complexity point of view. A practical integration of those Standards into popular web service frameworks is discussed in the last section of the thesis.

Abstrakt

Záměrem této diplomové práce je prozkoumání možností snížení komunikační a výpočetní zátěže protokolu XML, otestování popsaných metod redukujících tuto zátěž a jejich integraci do existujících technologií, které využívají protokol XML. V první části práce jsou popsány doporučení/standardy Fast Infoset a Efficient XML Interchange určené pro efektivní komunikaci pomocí protokolu XML. Druhá část práce je zaměřena na výkonnostní testy dostupných implementací těchto standardů z hlediska míry snížení komunikační zátěže a nároků na výpočetní výkon pro jejich zpracování. Poslední část práce se zabývá praktickou implementací těchto standardů v běžně používaných technologiích pro webové služby.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Již od doby prvních počítačů potřebovali tvůrci programů vyřešit jeden základní problém - jakým způsobem (de)serializovat vstupní/výstupní data? Hledání řešení tohoto problému zabralo mnoho času a jeho výsledkem je řada různých formátů, které jsou používány pro ukládání rozličných typů dat, od programů, přes textové dokumenty až po multimediální soubory. Většina těchto formátů ale sdílí jednu vlastnost, která je činí těžko použitelnými lidmi. Touto vlastností je binární formát ukládání dat, který přináší mnoho výhod jako jsou malá velikost a rychlé zpracování, ale z tohoto důvodu jsou soubory z velké části nečitelné a jejich případná interpretace člověkem je přinejmenším obtížná. Na druhou stranu soubory vytvořené člověkem, které jsou většinou textového charakteru, jsou velmi obtížně zpracovatelné počítačem, protože postrádají pevnou strukturu, která je potřebná pro jejich zpracování.

Z tohoto důvodu byla vytvořena řada jazyků, které definují strukturu zápisu dat, což umožňuje jejich strojové zpracování, ale zároveň jejich textový zápis umožňuje jejich přímou editaci člověkem. V závislosti na použití daného souboru se používá řada jazyků, které umožňují snadný zápis programů pro kompilátor (např. jazyk C/C++), tvorbu formátovaných textových dokumentů (např. LaTeX, ve kterém je napsána tato práce) nebo přenos dat mezi různými systémy pomocí značkovacích jazyků[1] (např. XML).

Cílem této práce je průzkum možností snížení náročnosti přenosu dat pomocí protokolů, které využívají pro přenos dat formát XML, se zvýšeným důrazem na velikost přenášených dat, a jejich otestování v reálných případech užití na platformách Microsoft Windows a GNU/Linux včetně jejich vzájemné interoperabilty.

1.1 Jazyk XML

Extensible Markup Language (XML) vznikl jako nástupce jazyka Standard Generalized Markup Language (SGML) ve druhé polovině 90. let 20. století jako jazyk vhodný pro tvorbu obsahu pro rychle rostoucí síť World Wide Web pod záštitou World Wide Web Consortium (W3C). Jazyk XML byl poprvé standardizován v roce 1998 a do dnešního dne v něm byly prováděny jen drobné změny a současná (5.) verze standardu je velmi rozšířená, podporovaná a doporučovaná k všeobecnému použití.

[1] Markup language

Dokument v jazyku XML se skládá z deklarací (typicky verze XML specifikace, kódování souboru, připojené XML Schema) a z elementů. Každý element má svoje jméno (tag) a může obsahovat atributy (pár název a hodnota) a další elementy. Dále mohou XML soubory obsahovat další informace jako jsou komentáře a sekce CDATA, které slouží k oddělení dat, která by mohla poškodit platnost souboru.

Ukázka jednoduchého XML souboru[2].

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
   </book>
   <book id="bk102">
      <author>Ralls, Kim</author>
      <title>Midnight Rain</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-12-16</publish_date>
      <description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</description>
   </book>
</catalog>
```

Z výše uvedené ukázky XML souboru je zřejmé, že XML obsahuje velké množství redundantních informací, které nejsou při strojovém zpracování souboru nutné - dlouhá (popisná) jména tagů, opakování jména tagu při jeho uzavření a vkládání konců řádků/tabelátorů/mezer pro vizualizaci struktury souboru. Již při zběžném pohledu je zřejmé, že tento formát může být velice neefektivní - například pro uložení hodnoty určující cenu knihy (pro zjednodušení můžeme předpokládat, že se jedná o 32 bitové desetinné číslo) budeme potřebovat uložit alespoň 4 znaky[3], což může odpovídat 4 bytům[4] v binární podobě, ale s rostoucí cenou se bude tato velikost zvětšovat. Navíc je nutné přičíst 15 dalších znaků pro obalení vlastní hodnoty do elementu a případné bílé znaky pro zachování struktury souboru.

Podobná situace jako u (desetinných) čísel bude nastávat i u dalších datových typů, které jsou vnitřně reprezentovány číslem, jako jsou například údaje o datu, času a v některých případech u výčtových datových typů. Velké režie na přenos se dočkáme také u datového

[2] Fragment pochází z <http://msdn.microsoft.com/en-us/library/ms762271%28v=vs.85%29.aspx>

[3] Předpokládejme, že minimální cena je 0.00.

[4] Při použití vhodného kódování, při použití kódování UTF-16 se bude jednat o 8 bytů.

typu boolean, který lze realizovat jedním bitem[5], obzvláště v případě, že jednotlivé hodnoty budeme reprezentovat jako `TRUE/FALSE` místo úspornějšího `0/1`. Naopak při předávání textových dat bude XML dosahovat stejné efektivity[6] jejich uložení jako při jejich reprezentaci v paměti.

Další nevýhodou XML oproti binárním souborům je rychlost jejich zpracování. Pro zpracování XML souborů se používají zejména 2 metody přístupu - proudové zpracování a objektový model dokumentu. Při použití proudového zpracování pomocí SAX[7] je dokument načítán jako proud vstupních elementů a pro každou načtenou součást souboru (nový element, atribut, vnitřní text elementu) je vyvolána nová událost, kterou aplikace podle potřeby obslouží nebo ignoruje. Tento přístup je vhodný zejména pro rozsáhlé soubory, které nemusíme udržovat načtené celé v paměti, ale není příliš vhodný pro úpravy dat a jiný než sekvenční přístup k datům. Druhým přístupem je vytvoření stromové struktury dokumentu v paměti pomocí DOM[8] a následné vyhledávání v této struktuře pomocí jazyka XPath, díky kterému lze dokument snadno prohledávat, ale za cenu velkého množství zabrané paměti. Oba zmiňované přístupy používají poměrně složitý parser, který je nutný pro zajištění jejich funkčnosti.

Přes výše zmíněné nevýhody je XML vhodné pro velké množství aplikací, protože kombinuje snadnou použitelnost pro člověka s možností dynamické změny struktury a nezávislostí na konkrétní aplikaci a platformě - formát XML je často používán pro výměnu dat mezi aplikacemi napříč platformami, pro volání webových služeb (např. v protokolu SOAP[9]) a je velice často používán pro konfigurační soubory aplikací. Pro XML hovoří i fakt, že jde o formát s podporou dalších technologií jako je již zmiňovaný dotazovací jazyk XPath, možnost transformací pomocí XSLT[10] nebo možnost definování struktury a omezení rozsahu hodnot pomocí XML Schema.

1.2 Optimalizace XML souborů

V určitých případech, jako jsou například konfigurační soubory, nám vyhovuje jistá výřečnost formátu XML, protože nám umožní předat uživateli dostatečný kontext o datech v souboru a nenutíme ho zbytečně studovat manuál nebo experimentovat. V takovémto případě je v podstatě lhostejné, jestli bude mít výsledný soubor o několik KB navíc a jeho zpracování bude trvat o několik ms déle. Existuje ale mnoho případů, kdy uživatel nepřijde do styku s generovanými daty, takže výřečnost XML představuje zbytečnou zátěž (zejména u malých souborů obsahujících jen několik málo hodnot může být velikost výsledného souboru až násobně větší než reálná velikost předávaných dat), ale zároveň nechceme přijít o flexibilitu, kterou nám XML poskytuje.

První optimalizací, která nám pomůže snížit velikost souboru je linearizace XML a zkrácení názvů tagů. Pro strojové zpracování je vedlejší jakou má soubor vizuální strukturu -

[5] Typická implementace využívá 1 byte.

[6] Opět bude záviset na kódování souboru, můžeme ale předpokládat, že bude program i výstupní soubor používat stejné kódování.

[7] Simple API for XML

[8] Document Object Model

[9] Simple Object Access Protocol

[10] Extensible Stylesheet Language Transformations

důležité je jenom pořadí jednotlivých elementů a jejich vzájemné zanoření. Zkracovaná tagů také nemá žádný vliv na vlastní data, ale pouze snižuje čitelnost dokumentu pro člověka[11]. Odstraněním těchto bílých znaků a zkrácením názvů tagů na pouze 1. písmeno z každého tagu došlo ke snížení velikosti ukázkového XML souboru z původních 726 bytů na 422 bytů, velikost souboru tedy klesla o 40%.

Další možností, jak zmenšit velikost XML souboru je jeho komprese pomocí bezztrátového kompresního algoritmu jakým je například Deflate. Tímto způsobem můžeme dosáhnout zmenšení velikosti souboru[12] za cenu zvýšené zátěže systému a stížené manipulace s daty. Výhodou tohoto řešení je, že zachovává (po dekomprimaci) původní strukturu souboru a lze jej tedy upravovat manuálně.

Závěrečnou možností, jak snížit velikost XML souboru je využití standardů odvozených od XML, které jsou binární reprezentací původních XML souborů. Jejich použitím tedy získáme soubory o menší velikosti a zvýšení výkonu (mimo jiné proto, že je parsován menší soubor), ale zároveň zůstane zachována možnost konvertovat tyto binární XML soubory do podoby "klasického" XML souboru se všemi jeho výhodami a nevýhodami. Prvním standardizovaným formátem pro binární XML je FastInfoset z roku 2005 (ITU) resp. 2007 (ISO). Novějším standardem je Efficient XML Interchange (EXI), který je v současné době ve stádiu doporučení W3C konsorcia.

V následujících kapitolách práce se zaměřím na fungování EXI a FastInfosetu, možnosti využití těchto standardů v reálném prostředí a na srovnání výkonu jednotlivých možností optimalizace XML souborů.

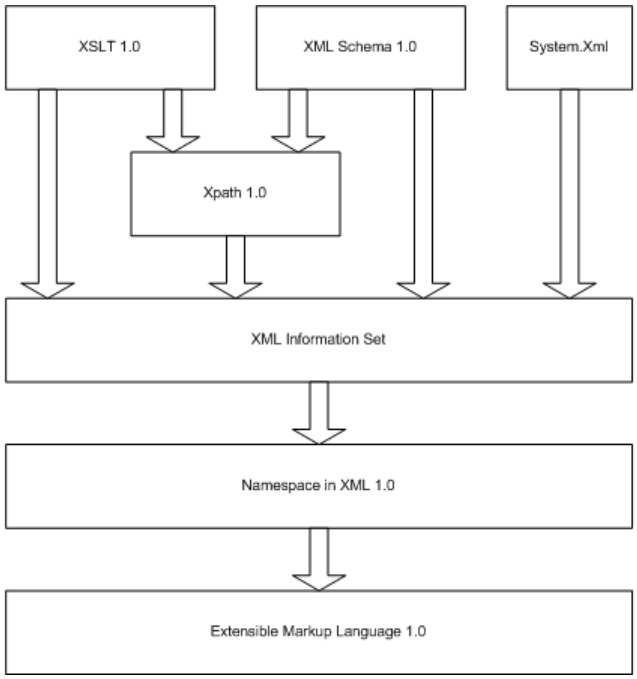
[11] Snížení čitelnosti (obfuskace) může být v některých případech dělána záměrně s účelem zvýšení obtížnosti reverse engineeringu aplikace nebo jejího komunikačního protokolu.

[12] Ve většině případů, v některých případech (zejména u malých souborů) ale může dojít i k navýšení velikosti.

Kapitola 2

XML Infoset

XML Infoset je Doporučením konsorcia W3C určeného pro reprezentaci XML dat v abstraktních datových strukturách, které slouží jako jednotné API[1] pro další technologie postavené na základě XML jako jsou XPath, XSLT nebo XSchema. XML Infoset je využíván i ve specifikacích Efficient XML Interchange a Fast Infoset, který je (zjednodušeně řečeno) jeho binární serializací. Z tohoto důvodu uvádím stručný přehled toho, co XML Infoset představuje, jaké nám nabízí možnosti a jakých omezení si musíme být vědomi při jeho použití.



Obrázek 2.1: Vztah mezi XML Infoset a dalšími technologiemi XML[2]

[1] Application Programming Interface
[2] Zdroj: <http://i.msdn.microsoft.com/dynimg/IC30283.gif>

Využití stejné datové struktury ve více technologiích přináší mnoho výhod jako jsou jednotné API, snadnější definice operací s odvoláním na strukturu Infosetu a abstrakce od konkrétní verze XML použité pro serializaci, případně vytvoření syntetického XML Infosetu, tj. Infosetu, který byl vytvořen bez existence příslušného XML dokumentu (může se jednat například o oddělenou část DOM[3]). Synteticky vytvořené Infosety nemusí být ve všech případech konzistentní/serializovatelné (problematické mohou být zejména dostupné/deklarované jmenné prostory).

XML Infoset je zaměřený na strukturu dat a jejich význam, ale nikoli na striktní reprezentaci XML dokumentu, ze kterého byl vytvořen. Z tohoto důvodu XML Infoset neobsahuje (z hlediska obsahu) bezvýznamné informace jako jsou způsob reprezentace prázdného elementu nebo textové oddělovače atributů. Při načtení XML dokumentu do XML Infosetu a jeho následnou zpětnou serializací (bez dalších úprav) můžeme získat rozdílné soubory, které budou obsahovat tyto drobné odlišnosti. Níže uvedené fragmenty jsou z pohledu XML Infosetu ekvivalentní.

```
<item></item>
<item />

<item attribute="value" />
<item attribute='value' />
```

2.1 Datový model

XML Infoset používá stromovou strukturu pro reprezentaci jednotlivých informačních součástí XML dokumentu jako jsou jednotlivé elementy, jejich atributy, jmenné prostory, komentáře a další[4]. Následuje popis nejdůležitějších informačních součástí a jejich nejdůležitějších vlastností.

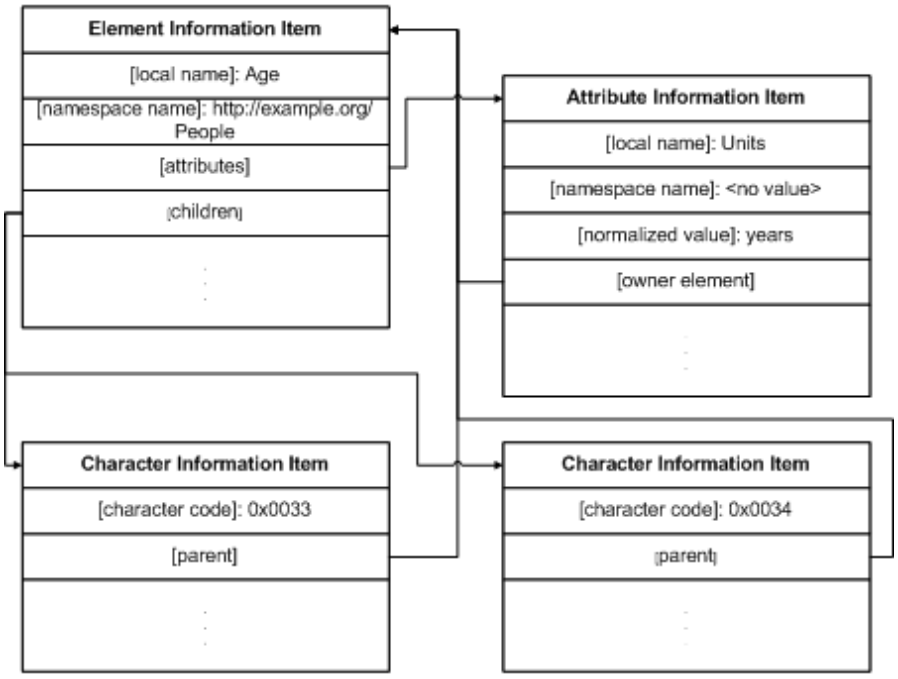
Na obrázcích 2.2 a 2.3 jsou znázorněny závislosti mezi níže zmiňovanými informačními součástmi, kde v prvním případě nejsou využívány jmenné prostory a v druhém případě je příklad o jmenné prostory rozšířen.

2.1.1 Document Information Item

Document Information Item je kořenovým prvkem datové struktury XML Infosetu - každý XML Infoset obsahuje právě jeden Document Information Item, všechny ostatní součásti XML Infosetu jsou dostupné přímo jako vlastnosti tohoto objektu, případně průchodem přes další objekty.

- **children** je uspořádaný seznam (pořadí je shodné s pořadím v XML dokumentu) podřízených informačních součástí. Součástí tohoto seznamu je kořenový element dokumentu, všechny instrukce pro zpracování a komentáře, které jsou umístěny mimo kořen-

[3] Document Object Model
[4] XML Infoset definuje celkem 11 různých informačních elementů.
[5] Zdroj: <http://i.msdn.microsoft.com/dynimg/IC164663.gif>
[6] Zdroj: <http://i.msdn.microsoft.com/dynimg/IC162104.gif>



Obrázek 2.2: Struktura datového modelu XML Infosetu bez použití jmenných prostorů[5]

Obrázek 2.3: Struktura datového modelu XML Infosetu při použití jmenných prostorů[6]

ový element. Mezi podřízené informační součásti je zařazena i deklarace typu dokumentu[7].

- **document element** je kořenovým elementem dokumentu.
- **version** určuje verzi XML dokumentu, pokud byla verze indikována v původním souboru. V případě, že verze indikována nebyla nemá tato vlastnost žádnou hodnotu[8].

2.1.2 Element Information Items

Element Information Item je abstraktní reprezentací jednoho elementu v XML dokumentu - včetně kořenového elementu, který je následně vložen jako vlastnost `document element` do Document Information Item. Všechny ostatní `Element Inforamation Items` jsou dostupné při rekurzivním procházení vlastnosti `children` tohoto elementu.

- **namespace name** obsahuje jméno jmenného prostoru, kterému tento element náleží. Pokud element neobsahuje specifikaci jmenného prostoru, pak tato vlastnost nemá hodnotu.
- **local name** obsahuje jméno elementu, bez jmenného prostoru a jeho oddělovače `":"`.
- **prefix** prefix jmenného prostoru, pokud kvalifikované jméno neobsahuje prefix, pak tato vlastnot nemá hodnotu.
- **children** uspořádaný seznam podřízených informačních součástí typů `element, processing instruction, unexpanded entity reference, character comment` v pořadí, ve kterém se objevují v původním dokumentu.
- **attributes** je neuspořádanou množinou `attribute` informačních součástí. Deklarace jmenných prostorů nejsou zahrnuty do této množiny. Pokud element nemá žádné atributy, potom bude tato množina prázdná.
- **parent** odkazuje na informační element, který obsahuje tento element ve vlastnosti children.

2.1.3 Attribute Information Items

Attribute Information Item reprezentuje atribut `elementu`, kterému náleží. `Attribute` je využíván pro hodnoty atributů i deklarace jmenných prostorů s tím rozdílem, že hodnoty atributů jsou v rodičovském elementu uloženy v množině `attributes` a deklarace jmenných prostorů jsou uloženy do množiny `namespace attributes`. Vlastnosti tohoto elementu obsahují již zmiňované vlastnosti `namespace name, local name` a `prefix` se stejným významem jako v předchozím případě. Dále obsahuje tyto vlastnosti:

[7] Document type declaration

[8] Ve specifikaci XML Infosetu jsou zmíněny dvě speciální hodnoty pro vlastnosti jednotlivých informačních součástí - `no value` (žádná hodnota) a `unknown` (neznámá hodnota), které jsou vzájemně různé a zároveň jsou odlišné od všech ostatních hodnot - zejména od prázdného stringu, prázdné množiny nebo seznamu. Tyto hodnoty nahrazují obecně používanou `NULL` hodnotu, které se specifikace z vyhýbá z důvodu větší obecnosti.

- **normalized value** je normalizovanou hodnotou[9] atributu.
- **specified** je příznakem, který indikuje, zda byla hodnota atributu přečtena z otevíracího tagu elementu nebo zda byla vložena výchozí hodnota specifikovaná v DTD.
- **attribute type** určuje typ atributu, pokud byl specifikován v DTD.
- **references** uspořádaný seznam elementů, na které tato hodnota odkazuje. Platí pouze v případě, že `attribute type` je `IDREF, IDREFS, ENTITY, ENTITIES` nebo `NOTATION`.
- **owner element** odkazuje na `Element Information Item`, kterému tento atribut náleží.

2.1.4 Character Information Items

Character Information Item reprezentuje jeden znak, který je obsažen v dokumentu. Každý znak je považován za samostatnou entititu, ale jednotlivé aplikace mohou používat shlukování znaků do bloků dle uvážení jejich tvůrce.

- **character code** je kódem znaku v kódování ISO 10646[10]
- **element content whitespace** je příznakem, zda se jedná bílý znak v obsahu elementu. Tato vlastnost je vynucena specifikací XML.
- **parent** odkaz na rodičovský element znaku.

2.1.5 Comment Information Items

Comment Information Items obsahují komentáře z XML dokumentu. Tato informační součást má pouze dvě vlastnosti:

- **content** obsahuje vlastní obsah komentáře.
- **parent** odkazuje na rodičovský element komentáře.

[9] Normalizace hodnoty je specifikována ve specifikaci příslušné verze XML. Při normalizaci dochází k nahrazení zástupných znaků znaky původními, nahrazení konců řádků a vrácení dalších změn, které byly provedeny před serializací hodnoty do XML.

[10] S omezením na platné znaky použitelné v XML.

Kapitola 3

Efficient XML Interchange

Efficient XML Interchange (EXI) je formátem pro velmi kompaktní a výkonnou reprezentaci XML Infosetu, který byl vyvinut pro snadné použití v širokém spektru aplikací. EXI výrazně snižuje nároky na velikost přenášených dat a zároveň zrychluje zpracování dat bez zvýšených nároků na systémové zdroje jako jsou výpočetní výkon a velikost paměti, výdrž na baterii nebo velikost zdrojového kódu [2]. Formát EXI je na rozdíl od formátu XML binární[1], takže není vhodný pro soubory, kde se očekává interakce s člověkem (konfigurační soubory), ale je vhodný například pro datovou výměnu mezi webovou službou a mobilním klientem (aplikací), kde může dojít k významnému snížení velikosti přenášených dat.

EXI je doporučením W3C[2] pro implementaci efektivního kódování XML dat, které bylo vytvořeno Efficient XML Interchange Working Group. Současná verze doporučení vychází z předchozího projektu společnosti AgileDelta - Efficient XML format[3].

V následujícím textu budu předpokládat, že zpracovávané XML soubory jsou používány pro výměnu dat mezi aplikacemi (webovými službami apod.) a nevyžadují žádné úpravy nebo kontroly od člověka, tudíž jsou vhodnými kandidáty na použití binárního přenosu dat, ale zároveň nechceme (nebo nemůžeme) používat vlastní binární formát[4]

3.1 Základní myšlenka

Základní myšlenkou, jak zmenšit velikost generovaného souboru je odstranění nepotřebných nebo redundantních informací. V případě XML souborů můžeme v mnoha případech "beztrestně"odstranit komentáře[5] a instrukce pro zpracování[6]. Komentáře se v XML objevují pouze zřídka, takže ve většině případů nedosáhneme žádného zmenšení, protože nemáme co odstranit. U instrukcí pro zpracování je situace podobná s jednou výjimkou. Běžný XML soubor by měl obsahovat informaci o verzi XML standardu a použitém kódování[7], kde obě

[1] V závislosti na nastavení může být navíc komprimován algoritmem Deflate

[2] W3C Recommendation

[3] Lightning-Fast Delivery of XML to More Devices in More Locations - <http://www.agiledelta.com/product_efx.html>

[4] Například z důvodu integrace do již existujících systémů nebo integrace s rozhraním od jiného dodavatele.

[5] Komentáře nejsou v XML určeny pro automatizované zpracování, tudíž je jejich existence v produkčních verzích aplikace beztak nežádoucí.

[6] Processing Instructions

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

tyto informace jsou z pohledu EXI irrelevatní, protože formát XML nebude použit a tudíž nemá smysl uvádět jeho verzi a informace o kódování je u binárního souboru také irrelevantní[8]. Ostatní informace pro zpracování mohou být pro koncovou aplikaci důležité nebo ne - zde pak záleží na rozhodnutí programátora, zda budou i ostatní informace pro zpracování odstraněny či nikoliv. Tímto způsobem jsme schopni omezit velikost dat, ze kterých bude vygenerován výsledný EXI soubor.

Další možností, jak dosáhnout zmenšení velikosti výsledného souboru je nahrazení tagů v souboru značkou události, například `StartElement` následovanou názvem elementu nebo odkazem do tabulky stringů[9] (bude popsána dále). Při uzavírání elementu událostí `EndElement` již není nutné uvádět jeho název resp. odkaz na jeho název, protože každý XML soubor se správnou strukturou[10] uzavírá tagy od posledního otevřeného (v ostatních případech by docházelo ke křížení tagů - některý z vnějších tagů by byl uzavřen dříve než všechny tagy jemu podřízené). Toto je tedy jedinou podmínkou pro použití EXI pro kódování XML dat - data musí dodržovat strukturu XML, žádné další požadavky na vstupní data (existence XML Schema, Document Type Definition atp.) nejsou kladeny.

Pro dosažení minimální velikosti souboru budeme potřebovat nahradit často málo efektivní reprezentaci dat stringem jejich binární podobou. Binární reprezentace je velice efektivní, zejména pro číselné datové typy (a datové typy od nich odvozené, jako jsou například datum a/nebo čas) a hodnoty typu boolean. Pro stringy bude (opět) použita tabulka stringů.

Po provedení výše popsaných operací vypadá ukázkový soubor následovně (v příkladu byly nahrazeny neplatné znaky zástupným znakem '?' a byly upraveny konce řádků, aby bylo možné příklad vytisknout na papír velikosti A4):

```
???catalog???book???id?bk101????author??Gambardella, Matthew?????title??XML
Developer's Guide?????genre?
Computer?????price??44.95????
publish_date??2000-10-01?????description?An in-depth look at creating appli
cations with XML.?????????bk102???Ralls, Kim????Midnight Rain??? Fantasy????
5.95????2000-12-16?????A former architect battles corporate zombies, an evil
sorceress, and her own childhood to become queen of the world.???
```

Jak je z příkladu patrné, došlo k nahrazení událostí pomocí kódů událostí a k nahrazení názvů elementů pomocí tabulky stringů, ale textový obsah všech elementů zůstal zachován a to i v případě, že se jedná o číselné datové typy. Toto je způsobeno nepřítomností schématu, který EXI informoval o datovém typu elementu, takže jsou všechny elementy považovány za textová data, a protože jsou vzájemně unikátní, tak jsou uloženy v plné formě.

Při použití XML Schema s dodatečnými informacemi je použito binárních formátů pro číselné datové typy. Také si můžeme povšimnout, že v souboru nejsou uloženy názvy jednotlivých tagů, ale jen odkazy do tabulky stringů[11]. Výsledný EXI soubor vypadá následujícím způsobem:

[8] Pro kódování datového typu `String` EXI používá vždy UNICODE, případně vlastní tabulku znaků

[9] V následujícím textu budu používat anglický termín "string" ve významu datového typu pro uchovávání posloupnosti znaků nebo ve významu samotné posloupnosti znaků. Budu také tento termín skloňovat pro lepší čitelnost textu.

[10] Well-formed XML

[11] Při použití XML Schema s předpokládá, že i dekodér na straně příjemce má toto schema k dispozici a názvy tagů následně snadno rekonstruuje z obou zdrojových souborů

```
?????bk101???Gambardella, Matthew????XML Developer's Guide???Computer?????#?
??????A?????<An in-depth look at creating applications with XML.?????bk102??
?Ralls, Kim????Midnight Rain??? Fantasy??????????????????????A former archite
ct battles corporate zombies, an evil sorceress, and her own childhood to be
come queen of the world.???
```

Příklad výše ale stále není minimální možnou EXI reprezentací původního XML a není tedy příliš vhodný pro reálné použití. Dalšího zmenšení lze dosáhnout při použití komprese dat, což je metoda vhodná spíše pro větší soubory (v některých případech může u malých souborů dojít ke zvětšení velikosti výsledného souboru) nebo použití možnosti Bit-packed, kde jsou jednotlivé hodnoty zarovnány na úrovni bitů namísto bytů jako v tomto případě. V případě komprese výsledného bude obsah zcela nečitelný, u Bit-Packed souborů je možné, že dojde (náhodně) k zarovnání na celé byty před začátkem stringu a některé stringy mohou být čitelné.

3.2 Datové typy v EXI

Klíčovým prvkem pro snížení velikosti výsledného souboru nebo datového proudu je využívání vhodných datových typů a jejich efektivní reprezentace v binární podobě. EXI používá pro reprezentaci dat několik základních typů, které jsou (s jedinou výjimkou) primitivními nebo knihovními datovými typy většiny běžně používaných programovacích jazyků. V závislosti na nastavení parametrů výstupu EXI mohou být jednotlivé hodnoty zarovnávány na celé byty (byte-packed) nebo na úrovni bitů (bit-packed).

3.2.1 Unsigned Integer

`Unsigned Integer` je reprezentován sekvencí bytů, kde poslední byte sekvence má MSB[12] nastaven na 0 - v ostatních případech je MSB nastaven na 1. V každém bytu je tedy obsaženo 7 bitů původního čísla a jeden bit je spotřebován na příznak konce sekvence. Původní číslo je při uložení rozděleno na 7-bitové bloky, které jsou uloženy od nejméně významného po nejvýznamnější. Toto uspořádání umožňuje snadnou rekonstrukci čísla při jeho načítání - je načten byte ze vstupu, 7 LSB[13] bitů je vymaskováno, posunuto o 7 * počet předchozích bytů a sečteno s předchozí hodnotou výsledného čísla[14]. Pokud je MSB bit nastaven na 1, tak je načten další bit, v opačném případě již bylo načteno celé číslo.

Každý EXI procesor musí být schopen zpracovat `Unsigned Integer` o velikosti 2^{32}, ale specifikace doporučuje, aby byl schopen zpracovat i čísla větší.

3.2.2 Boolean

Hodnoty typu `Boolean` mohou být v EXI reprezentovány dvěma způsoby - jako 1-bitový `Unsigned Integer`, kde hodnota 0 reprezentuje `FALSE` a hodnota 1 reprezentuje `TRUE` nebo

[12] Most significant bit
[13] Least Significant Bit
[14] Tato hodnota musí být na začátku nastavena na 0

jako 2-bitový `Unsigned Integer`, kde hodnota 0 reprezentuje `FALSE`, hodnota 1 reprezentuje `0`, hodnota 2 reprezentuje `TRUE` a hodnota 3 reprezentuje `1`. Výběr způsobu uložení je prováděn automaticky s ohledem na přiložené XML Schema.

3.2.3 Integer

`Integer` může být v EXI uložen 3-mi způsoby, které jsou použity v závislosti na omezeních hodnoty specifikovaném v přiloženém XML Schema. Pokud je počet hodnot omezen na 4096 nebo méně, potom je hodnota uložena jako `n` bitový `Unsigned Integer`, kde `n` je $\lceil log_2 m \rceil$ a `m` je počet hodnot.

Druhou možností uložení je využití typu `Unsigned Integer` v případě, že je rozsah hodnot zdola omezen hodnotou větší nebo rovnou 0. Ve všech ostatních případech je `Integer` uložen jako sekvence `Boolean` a `Unsigned Integer`, kde hodnota booleanu 0 označuje nezáporná čísla a hodnota 1 označuje záporná čísla a `Unsigned Integer` reprezentuje absolutní hodnotu čísla. V konečném důsledku je tedy `Integer` sekvencí 1 bitového `Unsigned Integeru` pro znaménko a n bitového `Unsigned Integeru` pro absolutní hodnotu čísla.

3.2.4 Float

`Float` je reprezentován velmi podobně jako v mnoha dalších případech - jako samostatné hodnoty exponentu a mantisy. Rozsah hodnot je vymezen maximální velikostí mantisy 64 bitů a maximálním exponentem 15 bitů - hodnoty, které jsou mimo tento rozsah nesmí být reprezentovány tímto typem. Dále je nutné zmínit, že hodnota exponentu -2^{14} je vyhrazena pro speciální použití - pokud je hodnota mantisy +/- 1, pak je výsledná hodnota považována za +/- nekonečno, v ostatních případech je hodnota čísla NaN[15]. Obě dvě složky typu `Float` jsou vnitřně reprezentovány jako `Integer`.

3.2.5 Decimal

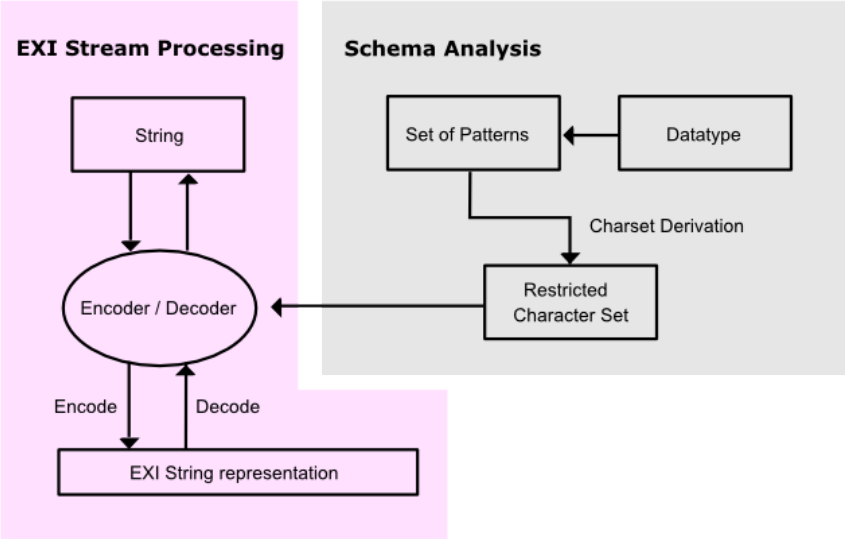
Decimal používá velmi jednoduchou reprezentaci. Celá část čísla je reprezentována jako `Integer`, desetinná část také (číslo je uloženo v opačném pořadí, aby zůstaly zachovány nuly za desetinnou čárkou - desetinná část z 19,0024 bude uložena jako 4200) a znaménko je reprezentováno jako `Boolean`.

3.2.6 String

Stringová data jsou EXI ukládána ve třech různých formách, v závislosti na místě výskytu stringu v rámci dokumentu a jeho případném omezení znakové sady. Omezenou znakovou sadu může mít pouze string, který reprezentuje obsah (vnitřní text elementu nebo hodnotu atributu). V nejjednodušším případě je string složen ze své délky (`n-bit Unsigned Integer`) a z vlastního obsahu v kódování Unicode. V případě, že byla původní XML data svázána pomocí XML Schema, ve kterém byl omezen rozsah hodnot pro datový typ odvozený od stringu, potom je daná hodnota kódována pomocí Restricted Character Set. Restricted Character Set

[15] Not a Number

vznikne enumerací všech povolených znaků pro danou hodnotu, setřízením těchto hodnot podle jejich indexu v Unicode a následným odkazováním na tyto indexy vyjádřené jako n-bit Unsigned Integer. Důležitým omezením je maximální počet 255 různých znaků a jejich přítomnost mezi Basic Multilingual Plane znaky Unicode[16]. Poslední možností jak zakódovat stringovou hodnotu je pomocí jejího indexu v tabulce stringů.



Obrázek 3.1: Schéma zpracování stringu[17]

Tabulka stringů je v EXI dynamicky vytvářena při kódování nebo dekódování souboru. Vzniká přidáváním jednotlivých načtených stringů[18] ze vstupního souboru v pořadí, ve kterém je soubor obsahuje. Při prvním výskytu nového stringu je jeho hodnota zapsána do souboru a zároveň je přidán do příslušné tabulky stringů. Při dalším použití stejného stringu je do souboru uložen jen jeho index v tabulce stirngů. Při generování tabulky je třeba striktně dodržovat pořadí, ve kterém jsou čteny, protože tabulky stringů nejsou součástí výsledného souboru - jsou po dokončení ukládání zahozeny a přijímající strana je rekonstruuje z dekódovaného souboru. Při (de)kódování jsou používány čtyři oddělené tabulky stringů pro URI, prefixy, QName a hodnoty atributů/vnitřní text elementu. V případě, že je XML svázáno pomocí XML Schema, tak je tabulka QName naplněna názvy elementů ze schema a do výsledného souboru jsou zapsány pouze indexy.

3.2.7 Date-Time

`Date-Time` je složeným datovým typem, který reprezentuje datum, čas a časovou zónu. Jednotlivé části jsou reprezentovány jako `n-bit Unsigned Integer`, pouze rok je reprezentován jako `Integer`, protože jeho hodnota vyjadřuje posun od roku 2000. Ostatní části jsou

[16] BMP znaky jsou v podstatě všechny znaky moderních jazyků. http://www.unicode.org/roadmaps/bmp/
[17] Zdroj: <http://www.w3.org/TR/2011/REC-exi-20110310/restrictedCharset.png>
[18] Vyjímku mohou tvořit stringy hodnot atributů a vnitřního textu elementů, pokud přesahují nastavenou délku nebo pokud vstupní data obsahují více rozdílných hodnot než je nastaveno. Obě dvě hodnoty se jsou nastaveny hlavičce EXI souboru.

sloučeny do celků MonthDay, Time, FractionalSecs a TimeZone - každá z těchto částí je reprezentována samostatně.

3.2.8 Binary

Binární data jsou v některých případech ukládána do XML souborů[19], ale před jejich uložením do XML je nutné zajistit jejich serializaci, aby nedošlo k porušení syntaxe XML. Jedním z běžně využívaných kódování binárních dat do podoby "prostého" textu (nejen pro využití v XML) je kódování Base64. Použitím kódování Base64 vzroste velikost přenášených dat o 25% - do každého bytu v Base64 je uloženo 6 bitů původních binárních dat. Toto řešení tedy není vhodné pro EXI, navíc je zbytečně složité. Protože je samotné EXI binární, tak nám nic nebrání uložit binární data do výstupního proudu v nezměněné podobě (po jednotlivých bytech[20]), kterému předchází jeho délka v bytech reprezentovaná jako `Unsigned Integer`.

3.2.9 List

`List` je sekvencí hodnot, který je předcházen jejich počtem (`Unsigned Integer`). Každá hodnota je uložena pomocí vlastního typu.

3.3 EXI Stream

EXI Stream je EXI Header následovaná EXI Body[21].

3.3.1 EXI Header

EXI Header slouží pro odlišení EXI Streamů od ostatních dat a zároveň je navržen tak, aby byl každý EXI Stream odmítnut při pokusu o zpracování v XML parseru. EXI Header má strukturu popsanou v následující tabulce.

[EXI Cookie]	Distinguishing Bits	Presence Bit for EXI Options	EXI Format Version	[EXI Options]	[Padding Bits]

EXI Cookie je sekvencí 4 bytů, které při použití kódování ASCII odpovídají znakům `"$EXI"` (v tomto pořadí). Cookie je nepovinnou součástí hlavičky - jejím účelem je snadnější rozpoznání platného EXI Streamu, její použití je doporučené.

[19] Například balíčky pro Microsoft SQL Server Integration Services (.dtsx) jsou XML soubory, které mohou obsahovat zkompilované části programů v Common Language Runtime (CLR).
[20] Ve specifikaci je použit termín oktet (octet)
[21] Definice z W3C Recommendation

Distinguishing Bits je sekvence dvou bitů - `0 1`. Jejím významem je jednoznačné rozlišení mezi EXI Streamem a XML dokumentem - přítomnost této sekvence v Headru je povinná. Důvodem, proč byla zvolena tato sekvence je fakt, že se jedná o minimální sekvenci, která nemůže být obsažena na začátku žádného platného XML dokumentu při použití nejrozšířenějších kódování (UTF-8, UTF-16, ISO 8859 a dalších), a proto by měl být EXI Stream okamžitě odmítnut každým standardním XML parserem. Tato sekvence může být použita pro jednoznačné rozlišení mezi XML a EXI v případě, že nedokážeme předem určit, v jakém formátu jsou příchozí data uložena.

EXI Format Version určuje verzi specifikace, podle které je následující stream vytvořen. Verze formátu obsahuje příznak testovací (Preview) verze specifikace a její číslo. EXI procesor musí být schopen přijmout EXI Stream stejné verze, jakou implementuje. V případě, že je stream jiné verze, není chování procesoru definováno.

EXI Options a Presence Bit for EXI Options jsou metadata, která specifikují parametry použité při vytváření výsledného EXI Streamu. Přítomnost v headru je nepovinná, ale předpokládá se, že tato metadata budou dostupná při dekódování streamu - mohou být přítomná v hlavičce, mohou být doručena dekodéru předem nebo může být o jejich zadání požádán uživatel. Ve všech případech je ale nutné zajistit jejich přítomnost, protože EXI neposkytuje žádný záložní mechanismus, kterým by bylo možné stream dekódovat bez přítomnosti EXI Options[22]. EXI Options jsou uloženy jako EXI Stream s výchozím nastavením s použitím XML Schema pro maximální snížení velikosti výstupního streamu. Pro jejich dekódování je nutné, aby měl EXI procesor k dispozici XML Schema pro EXI. Přehled EXI Options je v následující tabulce.

EXI Option	**Popis**	**Výchozí hodnota**
alignment	Určuje typ zarovnání hodnot (přítomnost paddingu), nesmí být použito zároveň s compression.	bit-packed
compression	Příznak komprese (bude popsána dále).	FALSE
strict	Vynucuje přesné dodržení připojeného XML Schema - jakékoli porušení schematu vyvolá výjimku.	FALSE
fragment	Příznak rozlišující, zda stream obsahuje celý dokument nebo jen jeho fragment[23].	FALSE

[22] Lze samozřejmě vyzkoušet výchozí nastavení, ale EXI jako takové neposkytuje žádnou deterministickou metodu na hledání těchto nastavení.

[23] Celým dokumentem je XML dokument, který obsahuje právě jeden kořenový element na rozdíl od fragmentu, který kořenový element nemusí obsahovat.

EXI Option	Popis	Výchozí hodnota
preserve	Sada příznaků, které umožňují zachovat komentáře, instrukce pro zpracování a další součásti XML, které nejsou nutné pro jeho korektní zpracování.	ALL FALSE
selfContained	Povoluje použití Self Contained elementů[24] v EXI Streamu. Nesmí být použito v kombinaci s compression, pre-compression a strict.	bez hodnoty
datatypeRepresentationMap	Vynucuje použití alternativních datových typů pro definované hodnoty.	bez hodnoty
blockSize	Velikost bloku při použití komprese.	1 000 000
valueMaxLength	Maximální délka hodnoty typu `String`, která bude přidána do string table.	bez omezení
valuePartitionCapacity	Maximální počet záznamů ve string table.	bez omezení
[user defined meta-data]	Uživatelská metadata, nesmí porušovat nebo upravovat nastavení EXI.	bez hodnoty

3.3.2 EXI Body

EXI Body je vlastní datová část streamu. Její obsah je řízen příslušnou gramatikou v závislosti na tom, zda byla původní data součástí dokumentu nebo jen jeho fragmentem a také na tom, jestli je pro daný dokument (fragment) k dispozici XML Schema. Podle těchto podmínek je vytvořena vhodná Built-in/Schema-informed document/fragment Grammar. Pomocí této gramatiky jsou následně generovány jednotlivé události (events), které slouží podobným způsobem jako tagy v XML. Jednotlivé události jsou do sebe zanořovány stejně jako v případě XML, nesmí docházet ke křížení uzavíracích událostí (mnoho událostí se musí ve streamu objevovat v párech - StartElement/EndElement). V EXI Streamech jsou podporovány jmenné prostory, pokud bylo jejich použití povoleno v EXI Options. Z důvodu větší kompaktnosti výsledného souboru jsou používány kódy pro jednotlivé události (Event Codes).

[24] Self Contained elementy jsou bez významové elementy, které mohou být čteny nezávisle od zbytku streamu a umožňují vyhledávání v rámci streamu.

3.4 Komprimace v EXI

V některých případech (zejména u větších souborů) můžeme dosáhnout snížení velikosti souboru jeho kompresí. Naivním přístupem ke kompresi souboru je výsledný soubor zkomprimovat jako celek. Toto řešení ale není v mnoha případech optimální, a proto EXI obsahuje zabudovaný algoritmus pro zefektivnění komprese založený na znalosti vnitřní struktury dat a jejich přeuspořádání do vhodnějšího pořadí. EXI navíc podporuje možnost pre-compression, která provede všechny kroky přípravy ke kompresi, ale vlastní komprimace není provedena, protože se předpokládá, že komprese bude provedena v rámci transportního protokolu. Tímto způsobem je zajišťena malá velikost přenášených dat a snížení výpočetní náročnosti odstraněním dvojité komprese, která typicky nepřináší další významné snížení velikosti souboru a tudíž zbytečně spotřebovává systémové zdroje.

3.4.1 Přehled

Prvním krokem, který EXI provádí před zahájením komprimace je rozdělení událostí v EXI streamu do bloků pevné délky[25]. Pevnou délkou je myšlen pevný počet EXI událostí, ale ne nutně shodná velikost bloku v bytech. Dalším krokem jsou jednotlivé kroky uspořádány do kanálů takovým způsobem, aby došlo k shluknutí podobných dat k sobě - kompresní algoritmy typicky vyhledávají opakující se bloky dat, které jsou následně komprimovány. Pokud jsou některé kanály výrazně kratší než jiné, mohou být spojeny do menšího počtu delších kanálů. Následně jsou kanály zkomprimovány[26] a přeposlány na výstup, v případě pre-compression jsou odeslány na výstup bez komprese.

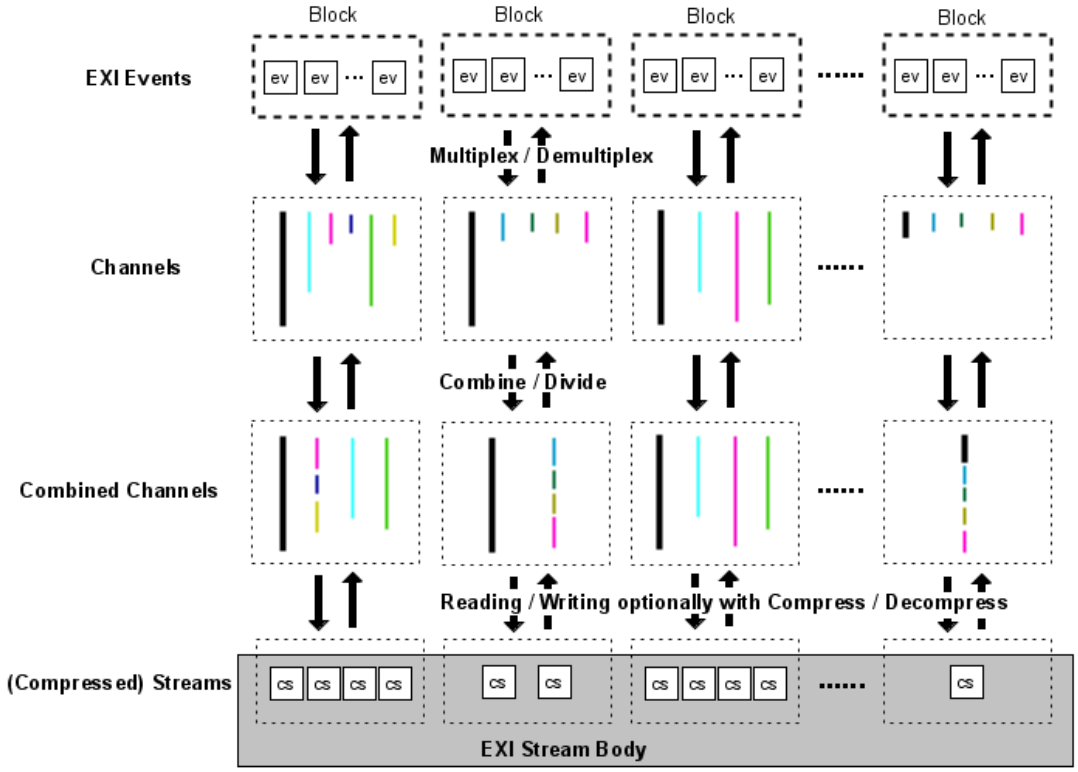
3.4.2 Kanály

Při seskupování "podobných"dat do jednotlivých EXI kanálů vycházeli vývojáři z jednoduché myšlenky, že obsah jednotlivých elementů a atributů bude v podobný, a proto seskupením jejich hodnot do jednoho kanálu získáme výhodu v následné komprimaci. Data z jednotlivých elementů jsou ukládána do dvou typů kanálů - strukturálního kanálu a do kanálu hodnot. Strukturální kanál obsahuje sekvenci událostí[28], které se v daném bloku vykytují (ve stejném pořadí) a kanál(y) hodnot obsahují data k těmto událostem. Hodnotových kanálů může být více, existuje právě jeden kanál pro každé qname[29].

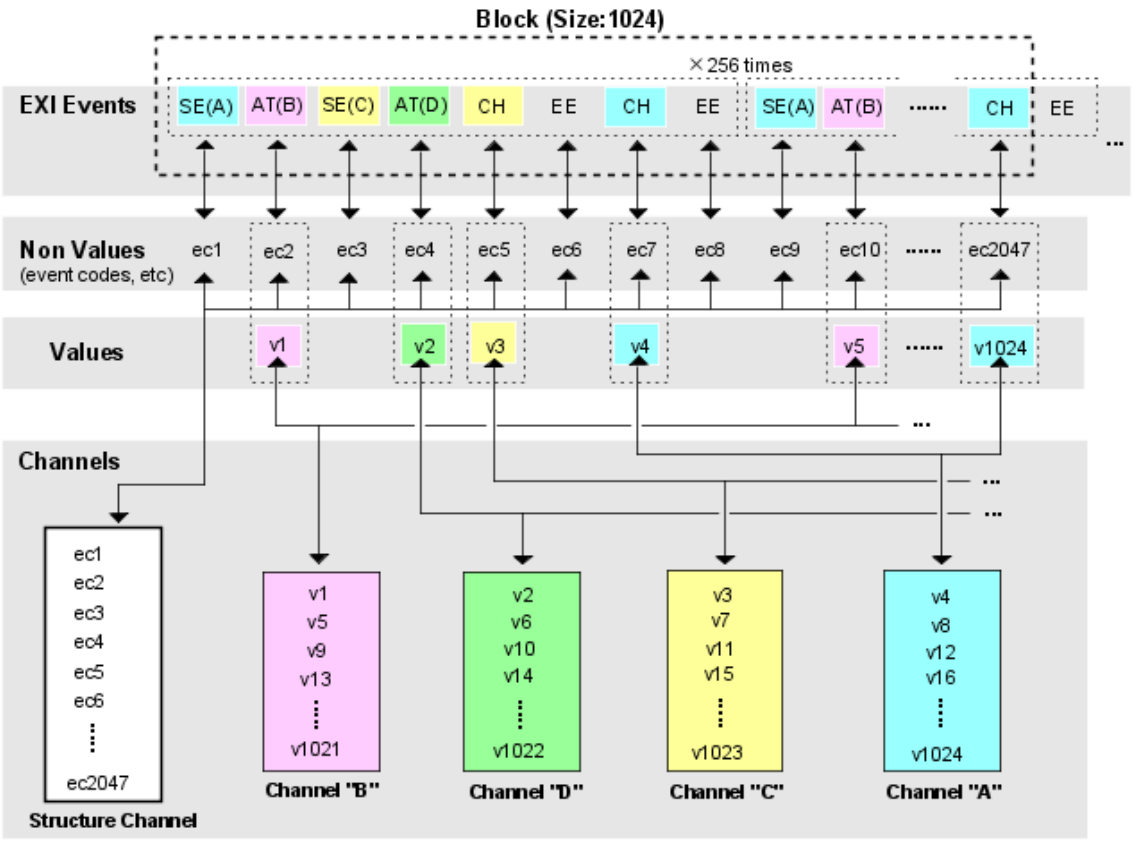
Pro seskupování kanálů je použito následujících pravidel:

- Pokud strukturální kanál obsahuje 100 událostí nebo méně, všechny kanály budou sloučeny do jednoho. Prvním kanálem bude strukturální, ostatní kanály budou uloženy v pořadí, ve kterém jsou odkazovány ze strukturálního kanálu.

[25] S výjimkovou posledního bloku, který může být kratší. Délka bloku je nastavena v hlavičce souboru.
[26] Jako kompresní algoritmus je využíván standard DEFLATE popsaný v IETF RFC 1951.
[27] Zdroj: <http://www.w3.org/TR/2011/REC-exi-20110310/compression.png>
[28] Navíc obsahuje data typů `xsi:type` a `xsi:nil`, protože ovlivňují gramatiku použitou pro generování souboru.
[29] Kvalifikované jméno elementu nebo atributu včetně jmenného prostoru.
[30] Zdroj: <http://www.w3.org/TR/2011/REC-exi-20110310/channels.png>



Obrázek 3.2: Kompresní schéma EXI[27]



Obrázek 3.3: EXI kanály[30]

- Pokud strukturální kanál obsahuje více než 100 událostí, bude uložen samostatně. Hodnotové kanály s nejvíce 100 hodnotami (pokud existují) budou sloučeny do jednoho kanálu (v pořadí, ve kterém jsou jejich hodnoty odkazovány ze strukturálního kanálu), který bude následovat strukturální kanál. Všechny zbývající kanály (pokud existují) budou následovat v pořadí, ve kterém jsou referencovány z kontrolního kanálu.

3.5 Dostupné implementace

EXI je otevřeným doporučením konsorcia W3C, které můžeme využít k vlastní implementaci nebo můžeme využít již existujících projektů, které z tohoto doporučení vycházejí. Následující seznam dostupných implementací je zveřejněn na webu W3C.

- **Efficient XML** je komerčním produktem společnosti AgileDelta Inc., ze kterého vychází velká část tohoto doporučení. K dispozici je komerční licence a 30 denní verze na vyzkoušení. Podporovanými platformami jsou Java (SE, EE i ME), Microsoft .NET (i Compact Framework) a C/C++ (verze pro Windows, Linux a další).

  <http://www.agiledelta.com/product_efxsdk.html>

- **EXIficient** je Open Source projektem uvolněným pod licencí General Public License v2 implementovaným v jazyku Java. Projekt vzniknul za podpory Siemens Corporate Technology.

  <http://exificient.sourceforge.net/>

- **OpenEXI** je Open Source projektem uvolněným pod Apache License Version 2.0 impletentovaným v jazyku Java. Projekt je veden společností Fujitsu.

  <http://openexi.sourceforge.net/>

- **EXIP** je Open Source projektem poskytovaný pod BSD 3-Clause License implementovaným v jazyce C. Projekt je veden skupinou EISLAB z Luleå University of Technology.

  <http://exip.sourceforge.net/>

Kapitola 4

Fast Infoset

Fast Infoset je doporučením/mezinárodním standardem[1] vytvořeným International Telecommunication Union (ITU) se záměrem vytvořit efektivní (ve smyslu velikosti a rychlosti zpracování) binární reprezentaci abstraktní datové struktury XML Infoset, která byla vytvořena konsorciem W3C, za použití kódování ASN.1[2]. Důvodem pro vytvoření tohoto standardu byla snaha o alternativní efektivní serializaci XML Infosetu pomocí již dostupných standardů.

Fast Infoset je přímým (mladším - pochází z roku 2005) konkurentem dříve popsaného formátu EXI se stejným zamýšleným použitím, tj. výměna dat mezi aplikacemi a webovými službami, kde nedochází k interakci člověka s daty, a proto můžeme využít výhody kompaktního binárního zápisu.

4.1 Základní myšlenka

Základní myšlenkou je využití zápisu strukturovaných dat pomocí předdefinovaných typů kódování ASN.1 pro serializaci jednotlivých částí XML Infosetu. Fast Infoset tedy definuje obecnou množinu ASN.1 typů, které jsou následně použity pro serializaci konkrétního Infosetu do výsledného souboru. Pro snížení velikosti výsledného souboru jsou ve Fast Infosetu používány ve velké míře tabulky referenčních stringů[3] (těchto tabulek je přítomno více, aby bylo možné zmenšit velikost jednotlivých odkazů; různé elementy Fast Infosetu využívají různé tabulky), omezené znakové sady a `encoding algorithm` pro ukládání určitých typů dat (typicky číselných a binárních v textové reprezentaci) v binární podobě.

4.2 Typy elementů Fast Infosetu

Specifikace Fast Infosetu používá 16 různých typů elementů, které reprezentuji jednotlivé informační elementy z XML Infosetu. Každý z těchto typů je definován pomocí ASN.1.

[1] Recommendation | International Standard

[2] Abstract Syntax Notation One

[3] Tyto stringové tabulky fungují velice podobně jako v případě EXI, a proto budou v následujícím textu zmiňovány bez dalšího popisu.

Následující výčet typů a jejich atributů není kompletní - zachycuje pouze nejdůležitější typy a jejich hlavní atributy. Kompletní výčet typů a jejich atributů lze nalézt ve specifikaci Fast Infosetu.

4.2.1 Document Type

XML dokument serializovaný pomocí Fast Infosetu začíná typem `Document`, stejně jako XML Infoset obsahuje ve svém kořeni `Document Information Item`. `Document Type` obsahuje na rozdíl od svého Infosetového protějšku více atributů jako jsou výchozí stringové tabulky (`prefixes, namespace-names, local-names`), dodatečné omezené znakové sady a kódovací algoritmy. Dále je obsažena informace o použitém kódování znaků - pokud není specifikováno, tak se předpokládá použití UTF-8, informaci o tom, zda je dokument `standalone` a verzi XML standardu, podle kterého byl dokument vytvořen. Hlavním atributem tohoto typu je ale beze sporu odkaz na vlastní data - tedy element typu `Element Type`, instrukce pro zpracování typu `ProcessingInstruction Type`, komentáře typu `Comment`. Níže je uvedena plná specifikace tohoto typu ze specifikace Fast Infosetu.

```
Document ::= SEQUENCE {
   additional-data SEQUENCE (SIZE(1..one-meg)) OF
      additional-datum SEQUENCE {
         id URI,
         data NonEmptyOctetString } OPTIONAL,
   initial-vocabulary SEQUENCE {
      external-vocabulary URI OPTIONAL,
      restricted-alphabets SEQUENCE (SIZE(1..256)) OF
         NonEmptyOctetString OPTIONAL,
      encoding-algorithms SEQUENCE (SIZE(1..256)) OF
         NonEmptyOctetString OPTIONAL,
      prefixes SEQUENCE (SIZE(1..one-meg)) OF
         NonEmptyOctetString OPTIONAL,
      namespace-names SEQUENCE (SIZE(1..one-meg)) OF
         NonEmptyOctetString OPTIONAL,
      local-names SEQUENCE (SIZE(1..one-meg)) OF
         NonEmptyOctetString OPTIONAL,
      other-ncnames SEQUENCE (SIZE(1..one-meg)) OF
         NonEmptyOctetString OPTIONAL,
      other-uris SEQUENCE (SIZE(1..one-meg)) OF
         NonEmptyOctetString OPTIONAL,
      attribute-values SEQUENCE (SIZE(1..one-meg)) OF
         EncodedCharacterString OPTIONAL,
      content-character-chunks SEQUENCE (SIZE(1..one-meg)) OF
         EncodedCharacterString OPTIONAL,
      other-strings SEQUENCE (SIZE(1..one-meg)) OF
         EncodedCharacterString OPTIONAL,
      element-name-surrogates SEQUENCE (SIZE(1..one-meg)) OF
         NameSurrogate OPTIONAL,
```

```
      attribute-name-surrogates SEQUENCE (SIZE(1..one-meg)) OF
         NameSurrogate OPTIONAL }
      (CONSTRAINED BY {
      -- If the initial-vocabulary component is present, at least
      -- one of its components shall be present -- }) OPTIONAL,
   notations SEQUENCE (SIZE(1..MAX)) OF
      Notation OPTIONAL,
   unparsed-entities SEQUENCE (SIZE(1..MAX)) OF
      UnparsedEntity OPTIONAL,
   character-encoding-scheme NonEmptyOctetString OPTIONAL,
   standalone BOOLEAN OPTIONAL,
   version NonIdentifyingStringOrIndex OPTIONAL
   -- OTHER STRING category --,
   children SEQUENCE (SIZE(0..MAX)) OF
      CHOICE {
         element Element,
         processing-instruction ProcessingInstruction,
         comment Comment,
         document-type-declaration DocumentTypeDeclaration }}
```

4.2.2 Element Type

`Element Type` je ekvivalentem typu `Element Information Item` XML Infosetu a v podstatě se tak jedná o reprezentaci jednoho tagu v "klasickém" XML. Element se skládá z očekávaných vlastností - jmenného prostoru, jména, atributů a obsahu, kterým mohou být další elementy, řetězcová data, komentáře nebo instrukce pro zpracování. `Element Type` je definován následovně:

```
Element ::= SEQUENCE {
   namespace-attributes SEQUENCE (SIZE(1..MAX)) OF
      NamespaceAttribute OPTIONAL,
   qualified-name QualifiedNameOrIndex
   attributes SEQUENCE (SIZE(1..MAX)) OF
      Attribute OPTIONAL,
   children SEQUENCE (SIZE(0..MAX)) OF
      CHOICE {
         element Element,
         processing-instruction ProcessingInstruction,
         unexpanded-entity-reference UnexpandedEntityReference,
         character-chunk CharacterChunk,
         comment Comment }}
```

4.2.3 Attribute Type

`Attribute Type` je určen pro serializaci XML Infoset typu `Attribute Information Item` a je tedy přímou reprezentací dodatečných atributů jednotlivých tagů. Ze své podstaty

se skládá ze jména a normalizované hodnoty, která nesmí být delší než 2^{32} znaků[4]. Obě tyto hodnoty mohou být reprezentovány literárními typy (QualifiedName/NonIdentifyingString) nebo indexy do příslušných tabulek stringů. Vlastní definice typu je následující:

```
Attribute ::= SEQUENCE {
   qualified-name QualifiedNameOrIndex
      -- ATTRIBUTE NAME category --,
   normalized-value NonIdentifyingStringOrIndex
      -- ATTRIBUTE VALUE category -- }
```

4.2.4 Character Chunk Type

`Character Chunk type` slouží k uchování `Character Information Item`, které pocházejí z jedné hodnoty[5] v jednom elementu. Délka hodnoty by neměla být rovna nule a musí být menší než 2^{32} znaků. Pro vlastní reprezentaci hodnoty je použit datový typ `NonIdentifying-StringOrIndex`, který umožňuje využití tabulek stringů a odkazování na již použité hodnoty.

```
CharacterChunk ::= SEQUENCE {
   character-codes NonIdentifyingStringOrIndex
      -- CONTENT CHARACTER CHUNK category -- }
```

4.2.5 Comment Type

Comment Type je reprezentací `Comment Information Item`. Jeho délka nesmí být větší než 2^{32} znaků.

```
Comment ::= SEQUENCE {
   content NonIdentifyingStringOrIndex -- OTHER STRING category --}
```

4.2.6 NonIdentifyingStringOrIndex Type

`NonIdentifyingStringOrIndex Type` je používán mnoha typy zmiňovanými výše pro reprezentaci řetězce znaků. Jeho hlavní výhodou je transparentní práce s tabulkami stringů - do vyšších typů je vždy propagována literálová hodnota a konkrétní reprezentace daného výskytu je skryta.

```
NonIdentifyingStringOrIndex ::= CHOICE {
   literal-character-string SEQUENCE {
      add-to-table BOOLEAN,
      character-string EncodedCharacterString },
      string-index INTEGER (0..one-meg) }
```

[4] Toto omezení zavádí definice ASN.1 z důvodu zvýšení efektivity kódování a zjednodušení implementace.

[5] V tomto kontextu výrazem "hodnota" není myšlena hodnota atributu nebo vnitřní text tagu, ale sekvence znaků, která je zpracovávána jako jeden string.

4.2.7 EncodedCharacterString Type

`EncodedCharacterString` je vlastní reprezentací řetězce znaků. Při tvorbě nové instance B`EncodedCharacterString` musí enkodér rozhodnout, jaké kódování bude pro daný string použito. Specifikace Fast Infosetu obsahuje dvě hlavní univerzálně použitelná kódování pro textová data (UTF-8 a UTF-16BE), dvě omezené znakové sady a sadu algoritmů pro efektivní kódování číselných a dalších běžně používaných hodnot. Pokud není délka výsledného bitového pole dělitelná osmi, potom je zarovnána pomocí bitů s hodnotou "1".

```
EncodedCharacterString ::= SEQUENCE {
   encoding-format CHOICE {
      utf-8 NULL,
      utf-16 NULL,
      restricted-alphabet INTEGER(1..256),
      encoding-algorithm INTEGER(1..256) },
   octets NonEmptyOctetString }
```

4.3 Restricted alphabet a Encoding algorithm

Fast Infoset využívá dvě metody pro snížení velikosti stringových dat pro hodnoty, které ze své podstaty mohou nabývat jen omezeného rozsahu hodnot[6].

První metodou je použití omezené znakové sady/abecedy (Restricted alphabet). Tato abeceda je uložena na začátku Fast Infoset dokumentu v elementu `Document Type`. Každá dodatečná abeceda je definována jako uspořádaná množina různých znaků ze sady ISO/IEC 10646. Každý Fast Infoset dokument obsahuje předdefinované abecedy `numeric` a `date and time` a dále může obsahovat až 254 dodatečných omezených znakových sad.

Druhou metodou použití binární reprezentace dat je `encoding algorithm`. Použití `encoding algorithm` umožňuje uložit binární data (reprezentovaná v hexadecimálním zápisu nebo v kódování base64), hodnoty typu boolean, UUID[7], CDATA (kódována pomocí UTF-8) a číselné datové typy[8]. Na `encoding algorithm` jsou ze strany specifikace Fast Infosetu kladeny tři základní podmínky:

- **URI** - Každý dodatečně definovaný `encoding algorithm` by měl obsahovat URI, aby jej bylo možné přidat do `encoding algortihm` tabulky.
- **Jednoznačnost** - Každý `encoding algorithm` musí přesně specifikovat na jaké řetězce hodnot je možné jej použít (včetně definic omezených znakových sad a délky - pokud taková omezení existují).
- **Reversibilita** Každý `encoding algorithm` musí být reversibilní - musí deterministickým způsobem produkovat výstup, ze kterého je možné deterministickým způsobem získat původní (nezměněný) výstup.

[6] Tyto hodnoty jsou například čísla, časová razítka nebo binární data

[7] Universally Unique IDentifier - IETF RFC4122

[8] Celočíselné short/int/long (16/32/64 bitové číslo v doplňkovém kódu) a čísla s plovoucí desetinnou čárkou float/double (32/64 bitové číslo dle IEEE 754)

4.4 Omezení

Fast Infoset je jednou z možných serializací XML Infosetu, který nabízí mnoho výhod, ale zároveň přichází s několika omezeními. XML Infoset musí splnit řadu podmínek, aby z něj bylo možné serializovat do Fast Infosetu. V první řadě musí být XML Infoset kompletní (vlastnost `all declarations processed` je nastavena na `true`), musí být konzistentní (z hlediska jmenných prostorů a referencí mezi atributy) a musí být korektním způsobem nastaveny příznaky u bílých znaků. Dále je nutné, aby normalizované hodnoty elementů typu atributů, komentářů a instrukcí pro zpracování byly kratší než 2^{32} znaků.

Fast Infoset neobsahuje kompletní datovou strukturu XML Infosetu, některé vlastnosti nejsou serializovány. Jedná se zejména o zpětné odkazy na rodičovské prvky v rámi stromové struktury (atribut `parent` u typu `element`, `attribute`, `comment` a dalších. Dále nejsou ukládány atributy `base URI` a některé další příznaky.

Většina omezení, které Fast Infoset předkládá se dotýkají pouze malého množství běžných XML dat, a proto nijak výrazně neomezují použitelnost Fast Infosetu v reálném provozu.

4.5 Dostupné implementace

Fast Infoset je mezinárodním standardem, který byl schválen ITU v květnu 2005 a organizací ISO o dva roky později, tedy v květnu 2007. Nejlépe podporovaným jazykem z hlediska Fast Infosetu je Java, dále jsou dostupné implementace pro jazyky C++ a C#.

- **Fast Infoset Project** je výchozí implementací Fast Infosetu pro jazyk Java verze 1.4 a vyšší, od verze 1.6 je součástí standardních knihoven. Dále je podpora Fast Infosetu dostupná ve výchozím stavu u serverů GlassFish verze 2 a Sun Java System Application Server Platform Edition verze 8.2 a 9.0. Implementace je dostupná v rámci Apache License verze 2.

  <https://fi.java.net/>

- **Liquid Fast Infoset** byl komerčním projektem společnosti Liquid Technologies Ltd, který byl uvolněn jako zkompilovaná knihovna a zdrojový kód pod licencí GNU Affero General Public License. Projekt je napsán pro Microsoft .NET Framework 2.0 a novější.

  <http://www.liquid-technologies.com/xml-compression.aspx>

- **FastInfoset.NET** je komečním projektem společnosti Noemax Technologies Ltd, který je nabízen pod proprietární licencí, pro vyzkoušení je nabízena 90 denní zkušební verze. Projekt je určen pro platformu Microsoft .NET Framework 2.0, podporována je i verze Compact Framework. Dále jsou podporovány další platformy společnosti Microsoft - Windows Phone 7.1 a 8.0, Silverlight 5 a Mono[9]. Noemax Technologies dále nabízí programy Fast Infoset Viewer a Fast Infoset Converter, které umožňují zobrazení Fast Infoset dokumentu ve formě XML respektive převod mezi XML a Fast Infoset dokumentem a obráceně.

  <http://www.noemax.com/products/fastinfoset/index.html>

[9] Mono je Open Source projektem zaměřeným na podporu běhu aplikací vytvořených pro Microsoft .NET Framework napříč platformami, které nejsou ze strany společnosti Microsoft podporovány.

- **Fast Infoset Tools for C/C++** je komerční produktem společnosti OSS Nokalva Inc. nabízený pod komerční licencí s možností vyzkoušení 30 denní trial verze. Produkt je určen pro jazyky C a C++ a je nabízen zkompilovaný pro platformy Microsoft Windows, GNU/Linux, Oracle Solaris a Symbian OS.

  <http://www.oss.com/xml/products/fi/fi-c.html>

Kapitola 5

Testovací metodika

Testovací metodika slouží ke stanovení postupů testování a popisuje co nejpřesněji podmínky, za kterých bylo testování prováděno. Definice jednoznačné testovací metodiky umožňuje v první řadě možnost opakovaného testování a přidávání dalších výsledků do relevantního srovnání, ukazuje případné omezující faktory, které se mohou v průběhu testování objevit a zároveň může sloužit pro hrubé srovnání výkonu na jiné hardwarové konfiguraci[1]. Testovací metodika také určuje jednotlivé případy použití/scénáře nasazení dané technologie, které jsou v jejím rámci testovány. V následujících sekcích se zaměřím na definici testovací metodiky pro porovnání výkonnosti Efficient XML Interchange, Fast Infosetu a kompresního algoritmu DEFLATE oproti nekomprimovanému/neoptimalizovanému formátu XML.

5.1 Testovací prostředí

Výkon jednotlivých implementací bude srovnáván na dvou laptopech, kde R500 bude sloužit k testům s operačním systémem z rodiny GNU/Linux a Microsoft Windows, druhý laptop - ProBook 6560 bude sloužit výhradně k testům na platformě Microsoft Windows. Všechny operační systémy použité pro testování jsou plně aktualizované, ale nejedná se o "čisté" instalace. Verze použitých knihoven (verze Javy a .NET Frameworku) budou specifikovány u jednotlivých konfigurací. V době běhu jednotlivých testů byly na testovacím stroji pozastaveny všechny úlohy, které nebyly nutné k úspěšnému dokončení jednotlivých testů.

5.1.1 Konfigurace R500

R500 je 6 let starý laptop Lenovo R500, který bude použit pro testy s operačním systémem GNU/Linux, použitá distribuce je Fedora 18 x86_64 s jádrem Linux 3.11.10 a OpenJDK 1.7.0_45. Druhý operační systém dostupný na tomto laptopu - Microsoft Windows 7 Professional ve 64-bitové verzi s .NET Framworkem 4.5 a Oracle Java Runtime Enviroment 1.7.0_45. Hardwarová konfigurace je následující:

[1] V omezené míře se můžeme pokusit přeškálovat výsledky testů s důrazem na to, že v systému může být skryto úzké hrdlo, které se v průběhu původního testování neprojevilo, ale může hrát významnou roli v omezení výkonu na výkonnějším hardwaru.

- **Procesor** Intel Core 2 Duo P8400 (Penrym, 2.26 GHz, 3MB L2 Cache, 2 járda (2 vlákna))
- **Operační paměť** 4 GB DDR3 1066 MHz (dva moduly)
- **Grafická karta** ATi Mobility Radeon 3470 128 MB
- **Pevný disk** Western Digital Scorpio Blue 1 TB (7200 otáček/min)

5.1.2 Konfigurace ProBook 6560

ProBook 6560 2 je 3 roky starý laptop HP ProBook 6560b s operačním systémem Microsoft Windows 7 Enterprise v 64-bitové verzi a bude použit pouze pro testy na platformě Microsoft Windows. Na tomto laptopu je nainstalován .NET Framework 4.5.1 a Oracle Java Runtime Enviroment 1.7.0_40. Jeho přesná hardwarová konfigurace je tato:

- **Procesor** Intel Core i5 2410M (Sandy Bridge, 2.3 - 2.9 GHz, 512 kB L2 Cache, 3 MB L3 Cache, 2 jádra s HyperThreadingem (4 vlákna))
- **Operační paměť** 8 GB DDR3 1600 MHz (dva moduly)
- **Grafická karta** AMD Mobility Radeon 6470M 512 MB
- **Pevný disk** Toshiba 500GB MK5061GSYN (7200 otáček/min)

5.1.3 Testované implementace

Pro testování byly zvoleny platformy Java a Microsoft .NET, které jsou velmi rozšířené, dostupné i na mobilních zařízeních a pro které jsou dostupné implementace Fast Infosetu a EXI. Testovány byly implementace zmiňované u jednotlivých algoritmů, následuje jejich seznam a použité verze.

5.1.3.1 EXI

Efficient XML Interchange bylo testováno pouze na platformě Java[2], protože se nepodařilo získat zkušební verzi[3] knihovny Efficient XML od společnosti AgileDelta. U EXI je důležitým parametrem typ použitého zarovnání nebo komprese při ukládání do souboru, proto je tato informace uváděna v závorkách u každého testu.

- **EXIficient**, Java, verze 0.9.2, dále označován jako EXIficient
- **OpenEXI**, Java, verze 0000.0002.0033.1, dále označovaná jako OpenEXI

[2] V průběhu tvorby této práce byla na stránkách projektu OpenEXI zveřejněna zpráva o vývoji implementace pro platformu MS .NET.

[3] Zkušební verze není volně dostupná ke stažení a na požadavek o poskytnutí zkušební verze jsem neobdržel žádnou odpověď.

5.1.3.2 Fast Infoset

- **Fast Infoset Project**, Java, součást frameworku Metro 2.3 a Project GlassFish, dále označován jako FI (Java) na platformě MS Windows a FI na platformě GNU/Linux[4]
- **Liquid Fast Infoset**, MS .NET, verze 1.0.6, dále označován jako FI (Liquid)
- **FastInfoset.NET**, MS .NET, verze 4.94.3512.0, dále označován jako FI (Noemax)[5]

5.2 Testovací scénáře

Pro otestování jednotlivých formátů a jejich implementací budou použity dva základní scénáře. V prvním scénáři se zaměřuji na výkon jednotlivých řešení z hlediska výpočetní náročnosti (je sledován celkový čas nutný pro dokončení dané operace) a na paměťovou náročnost daného řešení.

Časová náročnost bude měřena v kódu ukázkové aplikace pomocí integrovaných knihoven v Javě[6]/.NET Frameworku[7]. Před provedením měření času budou provedena neměřená "zahřívací" kola, jejichž účelem je zajistit načtení všech potřebných dat (knihoven, souborů) do operační paměti, aby byla eliminována přístupová doba pevného disku.

Paměťová náročnost bude bude sledována pomocí integrovaných knihoven pro sledování výkonu procesu. Z důvodu snížení vlivu přenosové rychlosti pevného disku budou testovací data nejdříve načtena do operační paměti a následně předána ke zpracování. Po dokončení transformace jsou data zapsána zpět do operační paměti. Tato měření jsou prováděna pomocí dvou metod. První metodou je sledování použité paměti procesu[8] a druhou metodou je sledování celkové alokované paměti procesem[9].

5.2.1 Náhrada XML souboru

Uvažovaným scénářem bude prostá serializace dat pro uchování na disku počítače nebo transport dat na fyzickém nosiči, případně pomocí protokolu pro výměnu dat. V reálném případě se může jednat například o import dat do aplikace nebo o uchovávání dokumentů vytvořených v dané aplikaci[10] nebo o volání webové služby[11].

[4] Ostatní testované implementace jsou určeny pro MS .NET Framework a nebyly testovány na GNU/Linuxu.

[5] Tato knihovna není z přiložena na CD se zdrojovými kódy, aby nebyly porušeny licenční podmínky vydavatele.

[6] System.nanoTime()

[7] System.Diagnostics.Stopwatch

[8] V Javě se jedná o rozdíl velikosti paměti Virtual Machine a velikosti volné paměti Virtual Machine, v .NETu lze tuto hodnotu získat vyčtením z Garbage Collectoru.

[9] Velikost paměti Virtual Machine v Javě, velikost `PrivateBytes64` z třídy Proce`ssInfo` v .NETu.

[10] Například open source kancelářský balík Libre Office používá pro ukládání dokumentů formát XML.

[11] Vzhledem k tomu, že při volání webové služby dochází k serializaci dat do XML nebo jiné formy pro transport, je tento případ v podstatě totožný s ukládáním souboru na disk za předpokladu, že v obou případech eliminujeme vlastnosti přenosového kanálu z důvodu opakovatelnosti měření, protože přístup k pevnému disku a vzdálenému síťovému zdroji mají podobnou charakteristiku a mohou velmi ovlivnit/zkreslit výsledky jednotlivých testů.

5.2.2 Vliv použití jmenných prostorů a XML Schema

Podpora jmenných prostorů je ve formátu velice důležitá, ale jejich podpora pravděpodobně nebude mít velký vliv na výkon a velikost souborů, a proto bude otestována pouze na konfiguraci ProBook 6560. Tento test zároveň slouží k vygenerování potřebných dat pro otestování interoperability implementací.

Podpora pro dodatečná metadata z externího zdroje (jako je například XML Schema) může v menší míře ovlivnit výkon a velikost výsledných souborů. Podpora pro XML Schema ve Fast Infosetu chybí úplně, v EXI bude otestována na konfiguraci ProBook 6560.

5.2.3 Interoperabilita jednotlivých implementací

Důležitou vlastností, kterou musí nově zaváděné formáty pro výměnu dat podporovat je existence více implementací, které jsou vzájemně kompatibilní. Z tohoto důvodu se zaměřuji kompatibilitu jednotlivých implementací pro obě zvažované platformy (GNU/Linux a Microsoft Windows). Pro otestování kompatibility jsou použity vygenerované soubory z prvního uváděného scénáře.

5.3 Testovací data

Vzhledem k faktu, že Fast Infoset i EXI používají podobné principy pro snížení velikosti přenášených dat[12] jsem se rozhodl použít sadu testovacích dat, které bude zaměřena na tyto optimalizované oblasti.

Sada dat svojí strukturou odpovídá typickému příkladu firemní CRM[13] aplikace, proto je jejím obsahem struktura "zákazník - objednávky". Data tedy obsahují typickou směs různých datových typů - stringy, celá čísla, desetinná čísla a další. Sada obsahuje 4 jednotlivé soubory s 1, 10, 100 a 10 000 zákazníky. Velikosti jednotlivých souborů byly zvoleny jako typické příklady použití 1 - detail pro jednoho zákazníka, 10 - stránkování v seznamu zákazníků, 100 - stránkování v seznamu zákazníků a 10 000 jako část importního souboru aplikace. Data z této sady budou označována jako Dxxxxx, kde xxxxx označuje velikost použité dávky.

Pro otestování výkonu při použití jmenných prostorů jsou výše zmiňovaná data vygenerována i s použitím tří jmenných prostorů, jeden pro každou entitu, která se v souboru vyskytuje.

Sada testovacích dat byla vygenerována z volně dostupné ukázkové databáze AdventureWorks2012 poskytované[14] společností Microsoft k otestování možností databázového serveru Microsoft SQL Server 2012.

[12] Tabulky stringů a binární reprezentace číselných a některých dalších dat.
[13] Customer Relationship Management
[14] Volně ke stažení na adrese: <http://msftdbprodsamples.codeplex.com/releases/view/93587>

Kapitola 6

Testy

V této kapitole jsou uvedeny výsledky testů, které byly provedeny pomocí testovací metodiky v předchozí kapitole. Výsledky testů jsou uvedeny v tabulkách a zobrazeny v grafech, které mají v některých případech omezenou zobrazovanou horní mez, aby byl patrný průběh grafu v oblasti našeho zájmu.

6.1 Velikost souborů

V mnoha případech může být hlavním kritériem pro výběr formátu dat velikost výsledného souboru po provedení převodu/komprese. Z tohoto hlediska vychází nejlépe EXI, který zejména při použití komprese dosahuje obdivuhodných výsledků.

V následujících tabulkách jsou uvedeny velikosti původních souborů a jejich velikosti po konverzi (všechny velikosti souborů jsou uváděny v kB). U souborů ve formátu XML jsou uváděny dvě velikosti souborů - verze označená jako "lineární"je XML, které neobsahuje konce řádků a odsazení tagů pro vytvoření vizuálního kontextu. Lineární XML je tedy proud XML tagů a jejich dat v jednom řádku. Naopak XML bez přídomku obsahuje konce řádků a odsazení a z tohoto důvodu je výrazně větší. Informační hodnota obou zmiňovaných souborů je totožná, ale již pouhou linearizací XML snížíme velikost původního souboru o zhruba jednu třetinu.

V této fázi byla objevena překvapující chyba v implementaci Fast Infosetu v Javě, kde tato konkrétní implementace neodstraňuje korektně všechny "bílé" znaky, a proto dosahuje výrazně horších výsledků než testované implementace pro platformu .NET. Ostatní algoritmy dosahují při použití stejných výchozích hodnot identických výsledků.

Pro EXI jsou uváděny hodnoty výsledků s výchozím zarovnáním bit-packed, výsledky označené písmenem "C" jsou při použití komprese[1].

6.2 Čas konverze

Druhým velmi důležitým parametrem jednotlivých algoritmů (a jejich implementací) je doba konverze souboru z XML do nového formátu. Pro tento test byl soubor načten do

[1] Kompresí je v tomto kontextu míněna interní komprese v rámci EXI pomocí algoritmu Deflate. Podrobnější popis je v sekci 3.4.

	XML	XML (lin.)	FI (Java)	FI (Noemax)	FI (Liquid)
D1	4.1	2.8	1.4	1.2	1.2
D10	37.5	25.9	8.3	6.6	6.6
D100	369.9	254.7	67.8	51.4	51.4
D1000	3585.4	2467.8	586.4	428.6	428.6
D10000	22603.9	15717.2	3582.5	2675.5	2675.5

Tabulka 6.1: Fast Infoset - velikosti souborů (kB)

	XML	XML (lin.)	OpenEXI	OpenEXI C	EXIfic.	EXIfic. C
D1	4.1	2.8	1.1	0.7	1.1	0.7
D10	37.5	25.9	4.8	2.5	4.8	2.5
D100	369.9	254.7	35.8	15.5	35.8	15.5
D1000	3585.4	2467.8	282.3	107.6	282.3	107.6
D10000	22603.9	15717.2	1721.5	616.9	1721.5	616.9

Tabulka 6.2: EXI - velikosti souborů (kB)

	XML (lin.)	FI (Java)	FI (Noemax)	FI (Liquid)
D1	68.29%	34.15%	29.27%	29.27%
D10	69.07%	22.13%	17.60%	17.60%
D100	68.86%	18.33%	13.90%	13.90%
D1000	68.83%	16.36%	11.95%	11.95%
D10000	69.53%	15.85%	11.84%	11.84%

Tabulka 6.3: Fast Infoset - kompresní poměr

	XML (lin.)	OpenEXI	OpenEXI (C)	EXIficient	EXIficient (C)
D1	68.29%	26.83%	17.07%	26.83%	17.07%
D10	69.07%	12.80%	6.67%	12.80%	6.67%
D100	68.86%	9.68%	4.19%	9.68%	4.19%
D1000	68.83%	7.87%	3.00%	7.87%	3.00%
D10000	69.53%	7.62%	2.73%	7.62%	2.73%

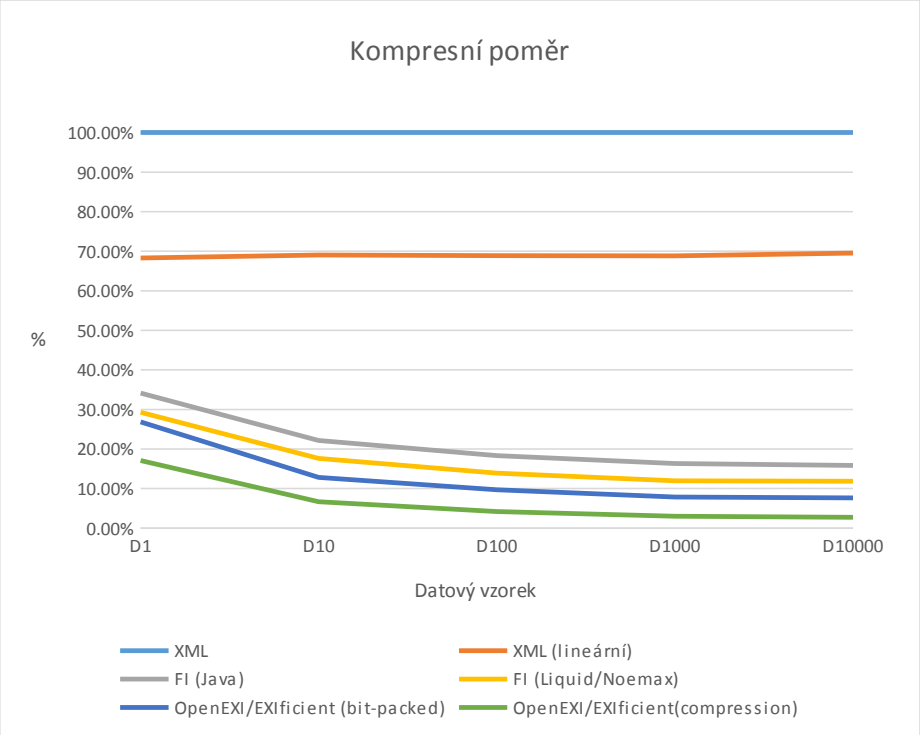
Tabulka 6.4: EXI - kompresní poměr

	FI (Java)	FI (Noemax)	FI (Liquid)
D1	50.00%	42.86%	42.86%
D10	32.05%	25.48%	25.48%
D100	26.62%	20.18%	20.18%
D1000	23.76%	17.37%	17.37%
D10000	22.79%	17.02%	17.02%

Tabulka 6.5: Fast Infoset - kompresní poměr proti lineárnímu XML

	OpenEXI	OpenEXI (C)	EXIficient	EXIficient (C)
D1	39.29%	25.00%	39.29%	25.00%
D10	18.53%	9.65%	18.53%	9.65%
D100	14.06%	6.09%	14.06%	6.09%
D1000	11.44%	4.36%	11.44%	4.36%
D10000	10.95%	3.92%	10.95%	3.92%

Tabulka 6.6: EXI - kompresní poměr proti lineárnímu XML



Obrázek 6.1: Kompresní poměry

operační paměti[2] a tento stream je následně předán k parsování a konverzi příslušnému algoritmu. Použitý parser se ve Fast Infosetu a EXI principiálně liší - Fast Infoset ze své podstaty používá DOM a je tedy výrazně paměťově náročnější než EXI, které využívá SAX. Pro porovnání s XML je XML soubor rozparsován do DOM a následně uložen.

Pro každou testovanou platformu jsou uváděny dvě sady výsledků encode a decode, kde encode je čas konverze z XML do Fast Infosetu/EXI a decode je čas převodu z Fast Infosetu/EXI do XML. Pro XML do XML bylo měření provedeno pouze jedenkrát, protože tato operace je obousměrně ekvivalentní.

Pro zvýšení relevance výsledků bylo po načtení souboru do paměti provedeno N iterací[3], které nebyly měřeny. Tyto iterace zajistí načtení potřebných knihoven do paměti a prodlouží běh programu z několika ms na řádově sekundy a dojde k stabilnějšímu přidělení procesorového času plánovačem operačního systému[4]. V reálném nasazení, například na webovém serveru jako výstupní enkodér pro komunikaci s webovou službou, lze očekávat velké množství volání těchto knihoven a tedy tato počáteční volání simulují delší běh aplikace. Po provedení těchto počátečních iterací byla provedena jedna měřená iterace. Pro každou instanci problému bylo provedeno měření pro ověření stability výsledku, v tabulce je uvedena průměrná hodnota z těchto měření.

Stabilita výsledků při měření času (i velikosti použité paměti) byla výrazně lepší s MS .NET frameworkem, který byl i výrazně (v některých případech i o řád) rychlejší než Java. Tento výsledek mne velice překvapil, a proto jsem tato měření opakoval, ale vzhledem k tomu, že se podobný výkonový odskok projevil na obou testovacích počítačích, tak tato měření odpovídají realitě. Možnou příčinou tohoto výkonového rozdílu je Just In Time compiler v Java Virtual Machine, který dynamicky určuje, který kód bude interpretován a který bude přeložen do nativního kódu platformy, na které je program spuštěn.

Tato sekce je dále rozdělena na jednotlivé podsekce podle testovacích počítačů a platforem a na jejím konci je krátké srovnání výsledků mezi GNU/Linuxem a MS Windows 7.

[2] V případě Javy do `ByteArrayInputStream`, v .NETu do `MemoryStream`.

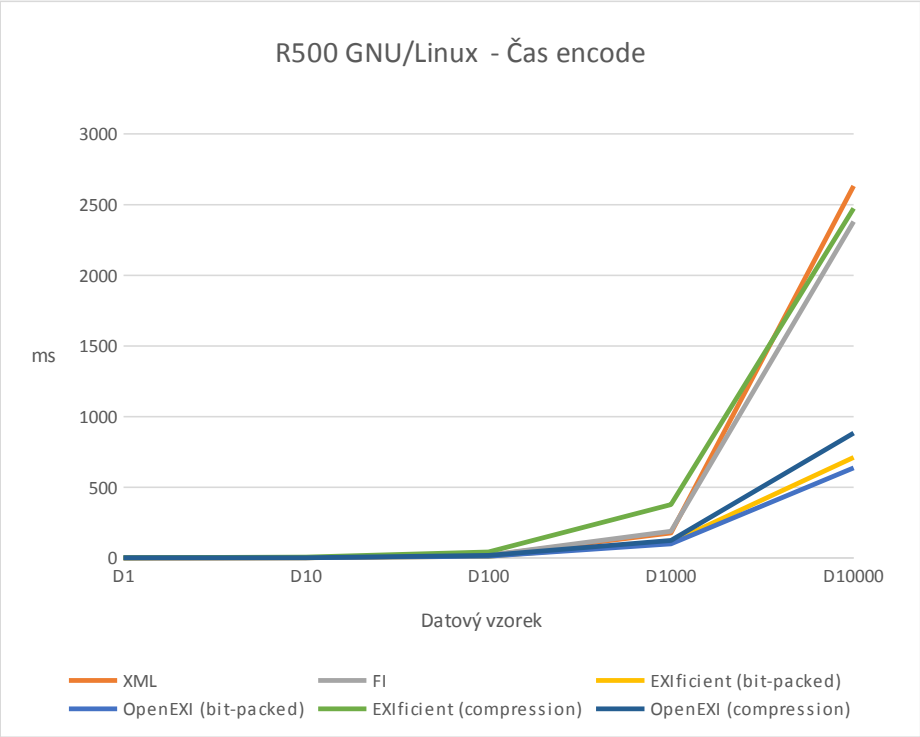
[3] 10 pro D10000, 100 pro D1000 a D100, 1000 pro D10 a D1

[4] GNU/Linux i MS Windows 7 používají preemptivní multitasking, a proto nelze zaručit, že během měření nedojde k přepnutí kontextu procesoru a tím ke zkreslení měřeného výsledku.

6.2.1 R500 - GNU/Linux

	D1	D10	D100	D1000	D10000
XML	0.73	2.39	18.85	176.08	2632.15
FI	0.75	2.49	19.79	189.35	2380.48
EXIficient (bit-packed)	0.49	1.62	12.33	109.67	710.77
OpenEXI (bit-packed)	0.80	2.24	11.98	98.79	637.50
EXIficient (compression)	1.32	5.83	42.25	377.72	2473.28
OpenEXI (compression)	1.07	2.45	17.57	123.07	882.80

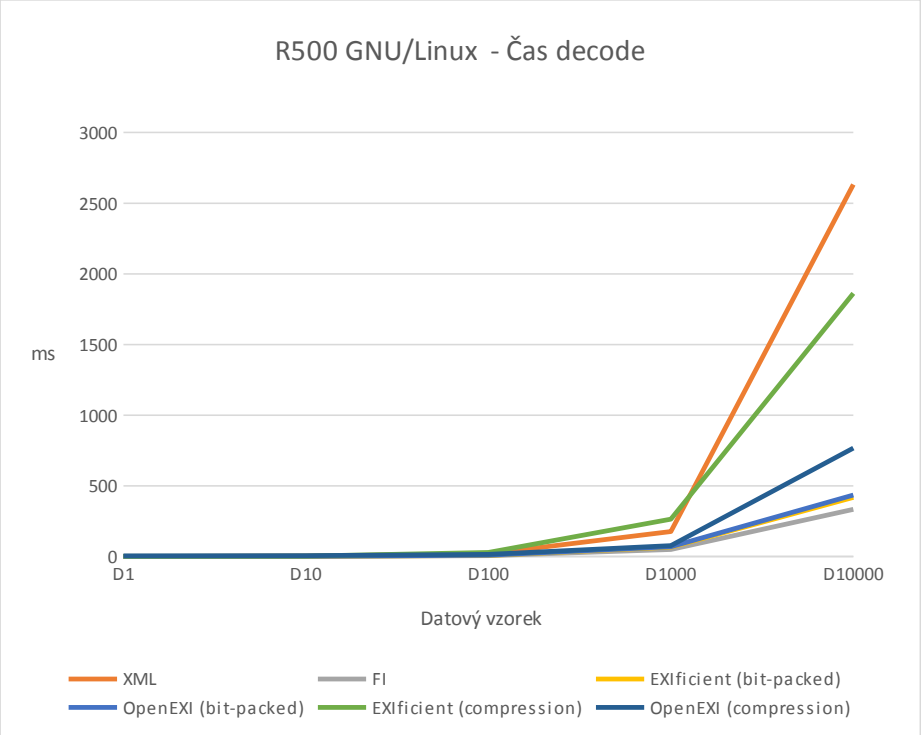
Tabulka 6.7: R500 - GNU/Linux čas encode (ms)



Obrázek 6.2: R500 - GNU/Linux čas encode

	D1	D10	D100	D1000	D10000
XML	0.73	2.39	18.85	176.08	2632.15
FI	0.30	0.84	6.07	49.90	333.46
EXIficient (bit-packed)	0.39	1.05	7.46	66.58	418.83
OpenEXI (bit-packed)	0.72	1.40	8.13	68.69	435.09
EXIficient (compression)	0.89	3.63	29.31	264.10	1862.28
OpenEXI (compression)	4.48	5.23	14.30	77.00	767.88

Tabulka 6.8: R500 - GNU/Linux čas decode (ms)

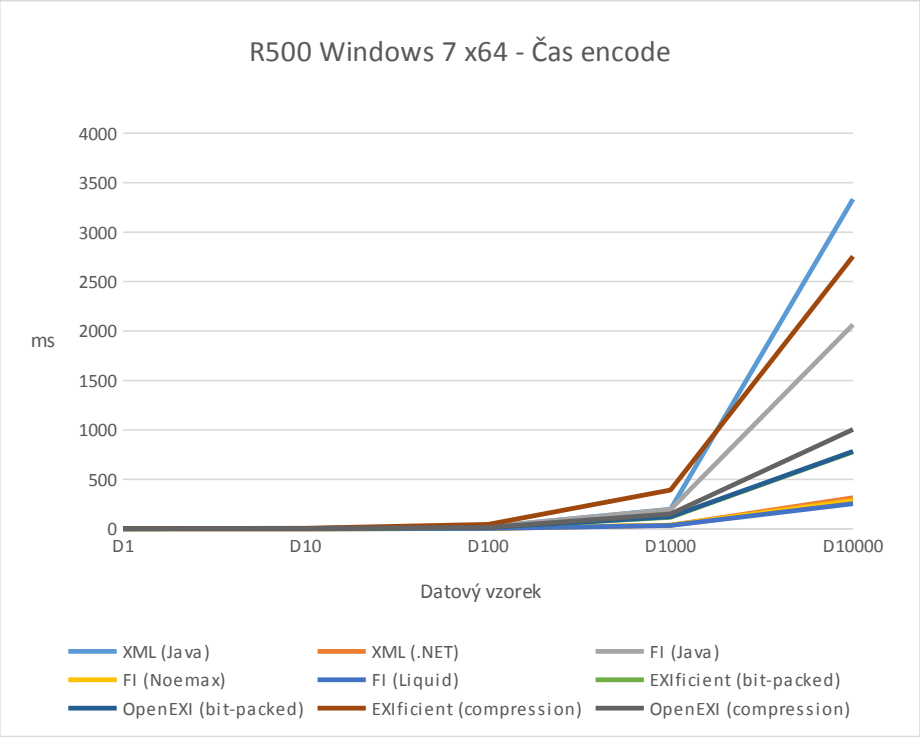


Obrázek 6.3: R500 - GNU/Linux čas decode

6.2.2 R500 - MS Windows 7

	D1	D10	D100	D1000	D10000
XML (Java)	0.82	2.46	20.44	197.31	3334.41
XML (.NET)	0.04	0.33	4.47	35.84	313.79
FI (Java)	1.00	3.00	20.71	195.28	2064.34
FI (Noemax)	0.04	0.29	3.75	38.46	287.06
FI (Liquid)	0.06	0.31	3.16	34.60	255.13
EXIficient (bit-packed)	0.70	1.69	13.85	116.36	777.67
OpenEXI (bit-packed)	1.45	2.90	13.32	121.68	782.08
EXIficient (compression)	1.40	6.03	44.13	391.02	2755.99
OpenEXI (compression)	1.33	2.86	15.09	150.09	1004.46

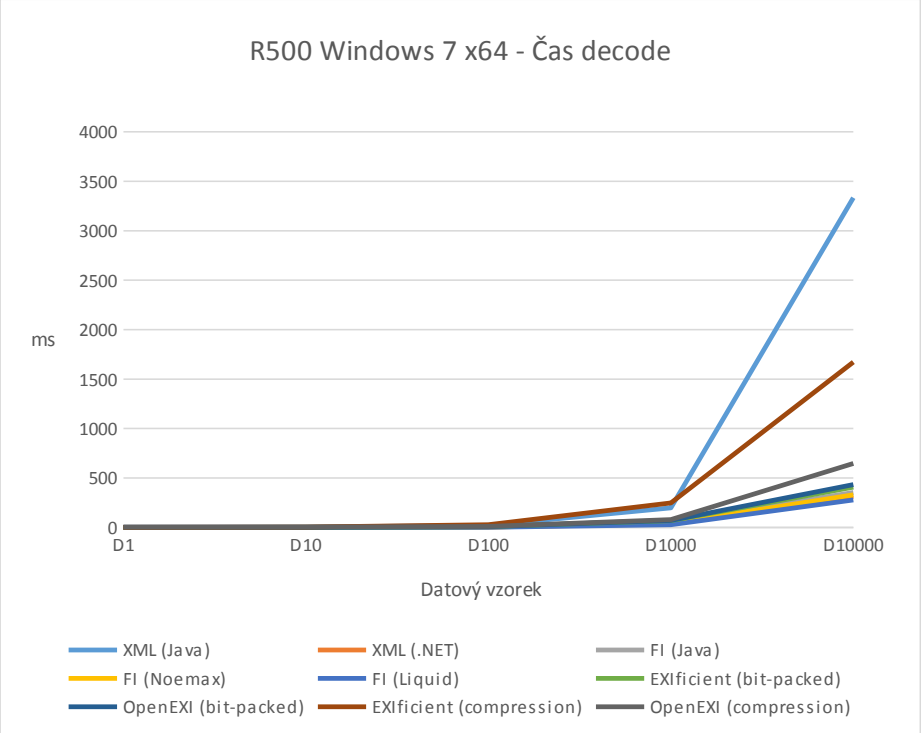
Tabulka 6.9: R500 - MS Windows 7 čas encode (ms)



Obrázek 6.4: R500 - MS Windows 7 čas encode

	D1	D10	D100	D1000	D10000
XML (Java)	0.82	2.46	20.44	197.31	3334.41
XML (.NET)	0.04	0.33	4.47	35.84	313.79
FI (Java)	0.35	0.86	5.78	55.52	352.32
FI (Noemax)	0.03	0.26	3.85	40.29	330.11
FI (Liquid)	0.03	0.24	2.35	27.58	276.84
EXIficient (bit-packed)	0.49	1.02	7.02	67.24	406.26
OpenEXI (bit-packed)	0.71	1.60	8.22	70.41	431.93
EXIficient (compression)	0.95	3.57	27.82	248.66	1672.77
OpenEXI (compression)	2.67	3.21	10.01	79.24	647.61

Tabulka 6.10: R500 - MS Windows 7 čas decode (ms)

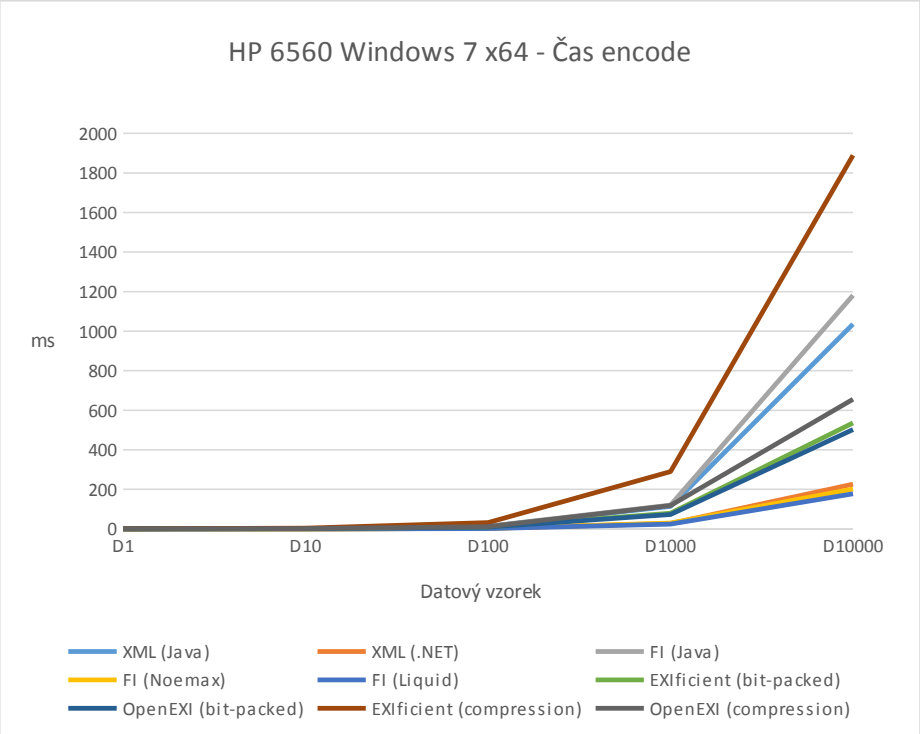


Obrázek 6.5: R500 - MS Windows 7 čas decode

6.2.3 ProBook 6560 - MS Windows 7

	D1	D10	D100	D1000	D10000
XML (Java)	0.68	1.91	12.99	113.37	1786.82
XML (.NET)	0.03	0.27	3.18	27.57	225.64
FI (Java)	0.95	2.15	13.45	120.58	1181.24
FI (Noemax)	0.03	0.22	2.66	30.25	201.25
FI (Liquid)	0.03	0.23	2.07	24.21	178.13
EXIficient (bit-packed)	0.50	1.35	9.02	79.80	536.41
OpenEXI (bit-packed)	0.83	1.69	9.56	73.05	502.24
EXIficient (compression)	1.21	4.25	32.32	289.81	1889.48
OpenEXI (compression)	1.13	2.01	11.09	118.37	655.37

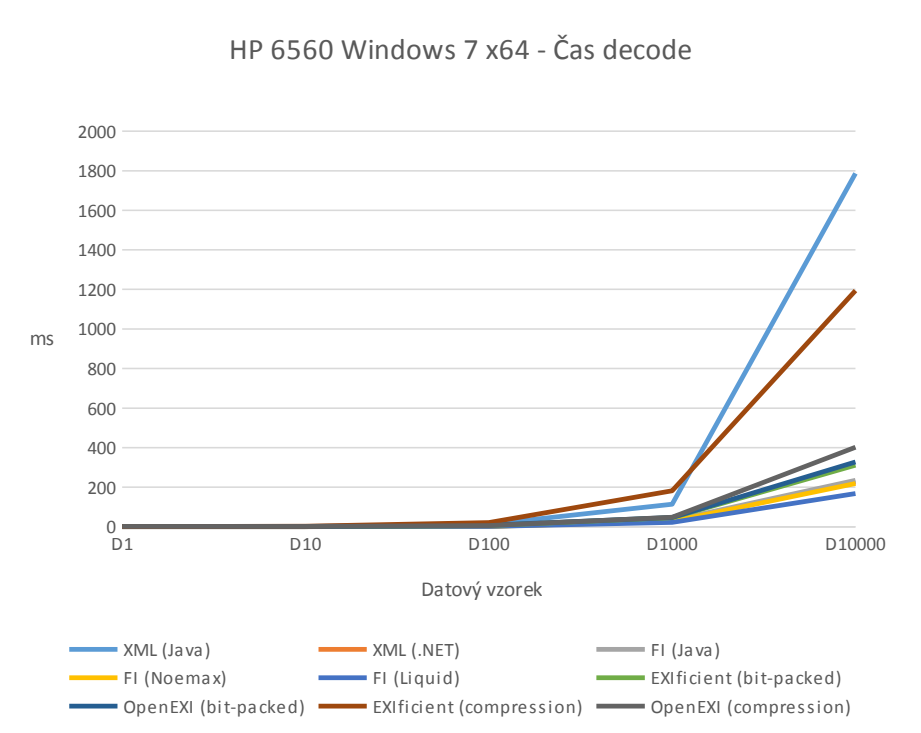
Tabulka 6.11: ProBook 6560 - MS Windows 7 čas encode (ms)



Obrázek 6.6: ProBook 6560 - MS Windows 7 čas encode

	D1	D10	D100	D1000	D10000
XML (Java)	0.68	1.91	12.99	113.37	1035.97
XML (.NET)	0.03	0.27	3.18	27.57	225.64
FI (Java)	0.24	0.59	3.72	31.55	234.70
FI (Noemax)	0.03	0.35	2.24	28.04	218.03
FI (Liquid)	0.02	0.17	1.77	22.10	167.32
EXIficient (bit-packed)	0.36	0.94	5.68	45.76	311.06
OpenEXI (bit-packed)	0.51	1.19	7.89	47.16	327.78
EXIficient (compression)	0.65	2.79	20.77	181.75	1194.20
OpenEXI (compression)	1.34	2.05	6.14	47.92	402.50

Tabulka 6.12: ProBook 6560 - MS Windows 7 čas decode (ms)



Obrázek 6.7: ProBook 6560 - MS Windows 7 čas decode

6.2.4 Srovnání výkonu

Vzhledem k tomu, že byla naměřena data v rozdílných operačních systémech se stejnou konfigurací, porovnání výkonu mezi stejnými implementacemi spuštěnými na různých platformách je zajímavou dodatečnou informací.

Z naměřených hodnot vyplývá, že rozdíl mezi platformou GNU/Linux a MS Windows 7 je dle očekávání malý. Zejména pro malé instance problému jsou rozdíly velmi blízko úrovně chyby měření, u větších instancí se projevuje rychlejší zpracování XML v Linuxu při konverzi z XML - zde je rozdíl ve výkonu více než 20%. Naopak při zpětné konverzi je výrazně výkonnější implementace ve Windows. V následujících tabulkách jsou uvedeny rozdíly jednotlivých měření, všechny hodnoty jsou v milisekundách.

Rozdíly ve výkonu jsou v následujících tabulkách vypočítány následovně:

- **Rozdíl** - (čas v Linuxu) - (čas ve Windows)
- **Rozdíl v %** - 1 - (čas ve Windows) / (čas v Linuxu)

	D1	**D10**	**D100**	**D1000**	**D10000**
XML (Java, Linux)	0.73	2.39	18.85	176.08	2632.15
XML (Java, Windows)	0.82	2.46	20.44	197.31	3334.41
Rozdíl	-0.09	-0.07	-1.59	-21.23	-702.26
Rozdíl v %	-12.33	-2.93	-8.44	-12.06	-26.68

Tabulka 6.13: Porovnání výkonu nativních parserů (ms)

	D1	**D10**	**D100**	**D1000**	**D10000**
FI (Java, Linux)	0.75	2.49	19.79	189.35	2380.48
FI (Java, Windows)	1.00	3.00	20.71	195.28	2064.34
Rozdíl	-0.25	-0.51	-0.92	-5.93	316.14
Rozdíl v %	-33.33	-20.48	-4.65	-3.13	13.28

Tabulka 6.14: Porovnání výkonu nativní implementace Fast Infosetu, encode (ms)

	D1	**D10**	**D100**	**D1000**	**D10000**
EXIficient (bit-packed, Linux)	0.49	1.62	12.33	109.67	710.77
EXIficient (bit-packed, Windows)	0.70	1.69	13.85	116.36	777.67
Rozdíl	-0.21	-0.07	-1.52	-6.69	-66.90
Rozdíl v %	-42.86	-4.32	-12.33	-6.10	-9.41

Tabulka 6.15: Porovnání výkonu EXIficient, encode, bit-packed (ms)

	D1	D10	D100	D1000	D10000
OpenEXI (bit-packed, Linux)	0.80	2.24	11.98	98.79	637.50
OpenEXI (bit-packed, Windows)	1.45	2.90	13.32	121.68	782.08
Rozdíl	-0.65	-0.66	-1.34	-22.89	-144.58
Rozdíl v %	-81.25	-29.46	-11.19	-23.17	-22.68

Tabulka 6.16: Porovnání výkonu OpenEXI, encode, bit-packed (ms)

	D1	D10	D100	D1000	D10000
EXIficient (compression, Linux)	1.32	5.83	42.25	377.72	2473.28
EXIficient (compression, Windows)	1.40	6.03	44.13	391.02	2755.99
Rozdíl	-0.08	-0.20	-1.88	-13.3	-282.71
Rozdíl v %	-6.06	-3.43	-4.45	-3.52	-11.43

Tabulka 6.17: Porovnání výkonu EXIficient, encode, compression (ms)

	D1	D10	D100	D1000	D10000
OpenEXI (compression, Linux)	1.07	2.45	17.57	123.07	882.80
OpenEXI (compression, Windows)	1.33	2.86	15.09	150.09	1004.46
Rozdíl	-0.26	-0.41	2.48	-27.02	-121.66
Rozdíl v %	-24.30	-16.73	14.11	-21.95	-13.78

Tabulka 6.18: Porovnání výkonu OpenEXI, encode, compression (ms)

	D1	D10	D100	D1000	D10000
FI (Java, Linux)	0.30	0.84	6.07	49.90	333.46
FI (Java, Windows)	0.35	0.86	5.78	55.52	352.32
Rozdíl	-0.05	-0.02	0.29	-5.62	-18.86
Rozdíl v %	-16.67	-2.38	4.78	-11.26	-5.66

Tabulka 6.19: Porovnání výkonu nativní implementace Fast Infosetu, decode (ms)

	D1	D10	D100	D1000	D10000
EXIficient (bit-packed, Linux)	0.39	1.05	7.46	66.58	418.83
EXIficient (bit-packed, Windows)	0.49	1.02	7.02	67.24	406.26
Rozdíl	-0.10	0.03	0.44	-0.66	12.57
Rozdíl v %	-25.64	2.86	5.90	-0.99	3.00

Tabulka 6.20: Porovnání výkonu EXIficient, decode, bit-packed (ms)

	D1	D10	D100	D1000	D10000
OpenEXI (bit-packed, Linux)	0.72	1.40	8.13	68.69	435.09
OpenEXI (bit-packed, Windows)	0.71	1.60	8.22	70.41	431.93
Rozdíl	0.01	-0.20	-0.09	-1.72	3.16
Rozdíl v %	1.39	-14.29	-1.11	-2.50	0.73

Tabulka 6.21: Porovnání výkonu OpenEXI, decode, bit-packed (ms)

	D1	D10	D100	D1000	D10000
EXIficient (compression, Linux)	0.89	3.63	29.31	264.10	1862.28
EXIficient (compression, Windows)	0.95	3.57	27.82	248.66	1672.77
Rozdíl	-0.06	0.06	1.49	15.44	189.51
Rozdíl v %	-6.74	1.65	5.08	5.85	10.18

Tabulka 6.22: Porovnání výkonu EXIficient, decode, compression (ms)

	D1	D10	D100	D1000	D10000
OpenEXI (compression, Linux)	4.48	5.23	14.3	77.00	767.88
OpenEXI (compression, Windows)	2.67	3.21	10.01	79.24	647.61
Rozdíl	1.81	2.02	4.29	-2.24	120.27
Rozdíl v %	40.40	38.62	30.00	-2.91	15.66

Tabulka 6.23: Porovnání výkonu OpenEXI, decode, compression (ms)

6.3 Paměťové nároky

Posledním sledovaným parametrem při konverzi souborů bylo množství spotřebované paměti. Toto měření bylo prováděno nad dvěma veličinami - bylo měřeno množství paměti alokované na heapu a celkové množství paměti alokované procesem. obě hodnoty byly vyčítány pomocí interních diagnostických nástrojů obou platforem[5].

Z hlediska paměťových nároků byly horší výsledky naměřeny u implementací v Javě, kde minimální hodnota alokované paměti JVM neklesá pod automaticky určenou hodnotu a v některých případech byly výsledky měření nestabilní, proto je v těchto situacích použita průměrná hodnota z jednotlivých měření.

V tomto testu nejsou uváděny naměřené hodnoty pro konfiguraci ProBook 6560, protože naměřené hodnoty jsou téměř totožné. Toto vychází z použití stejné platformy MS Windows 7 x64 a stejných major verzí .NET Frameworku a Java Runtime Enviroment. Jediným nalezeným závažnějším rozdílem je zvýšení limitu minimální alokované paměti pro proces v Javě na ProBooku 6560 na cca 125MB z cca 61MB na R500. Tento rozdíl je s největší pravděpodobností způsoben dvojnásobným množství operační paměti v ProBooku 6560 oproti R500[6]. Podrobné výsledky testu jsou k nahlédnutí na přiloženém CD v souboru se souhrnými výsledky testů.

6.3.1 R500 - GNU/Linux paměťové nároky

	D1	D10	D100	D1000	D10000
XML	3337	4447	14821	54546	281922
FI	1732	2542	12734	44952	231948
EXIficient (bit-packed)	5417	6173	10927	30481	135274
OpenEXI (bit-packed)	6225	6457	9320	22561	56979
EXIficient (compression)	5775	6186	12500	22112	94962
OpenEXI (compression)	6474	6591	9730	17520	77925

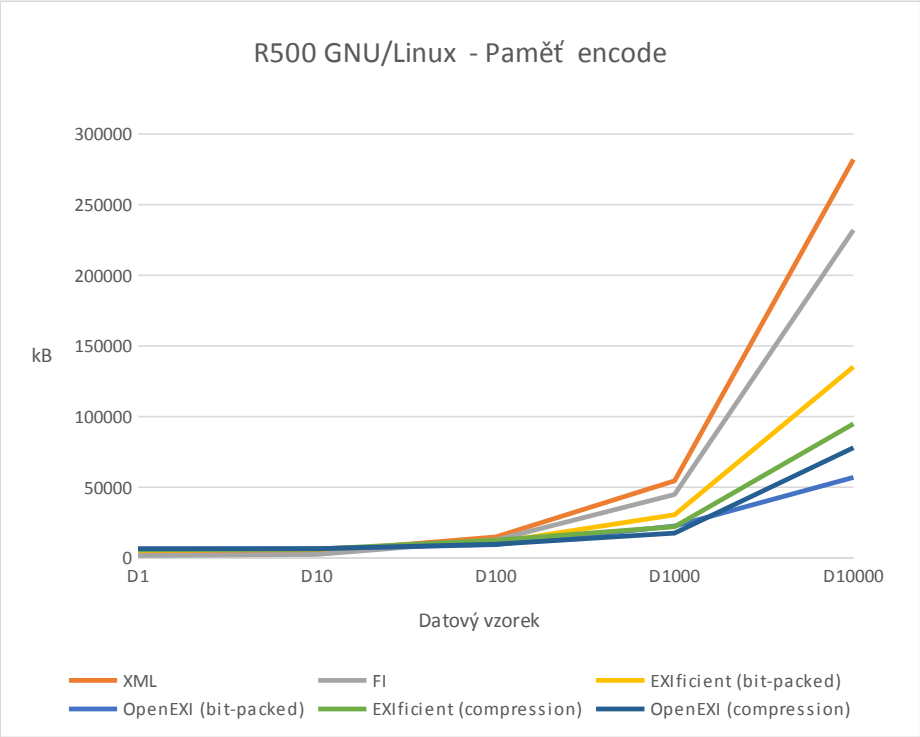
Tabulka 6.24: R500 - GNU/Linux paměťové nároky encode (kB)

	D1	D10	D100	D1000	D10000
XML	59904	59904	59904	120320	460800
FI	59904	59904	59904	118784	437248
EXIficient (bit-packed)	59904	59904	59904	75776	166400
OpenEXI (bit-packed)	59904	59904	59904	59904	94720
EXIficient (compression)	59904	59904	59904	75776	219136
OpenEXI (compression)	59904	59904	59904	75776	157696

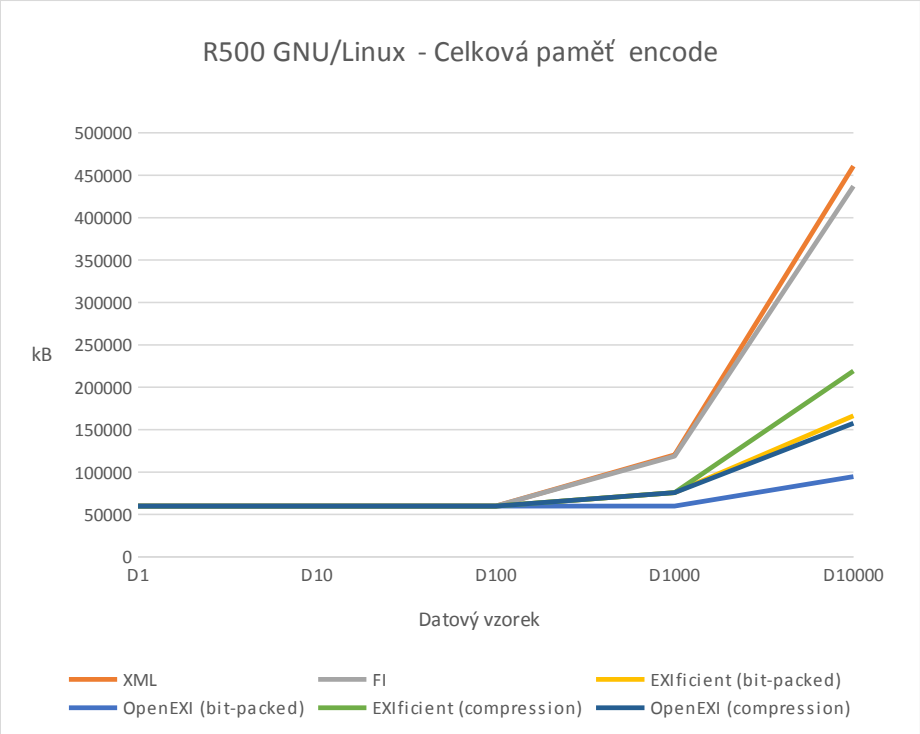
Tabulka 6.25: R500 - GNU/Linux celkové paměťové nároky encode (kB)

[5] V Javě se jedné o knihovnu Runtime, která poskytuje informace o stavu Java Virtual Machine, na které je proces spuštěn. V .NETu byly poouženy třídy ProcessInfo a GC (Garbage Collector), které poskytují obdobné informace jako třída Runtime v Javě.

[6] ProBook 6560 má k dispozici 8 GB RAM, R500 pouze 4 GB.



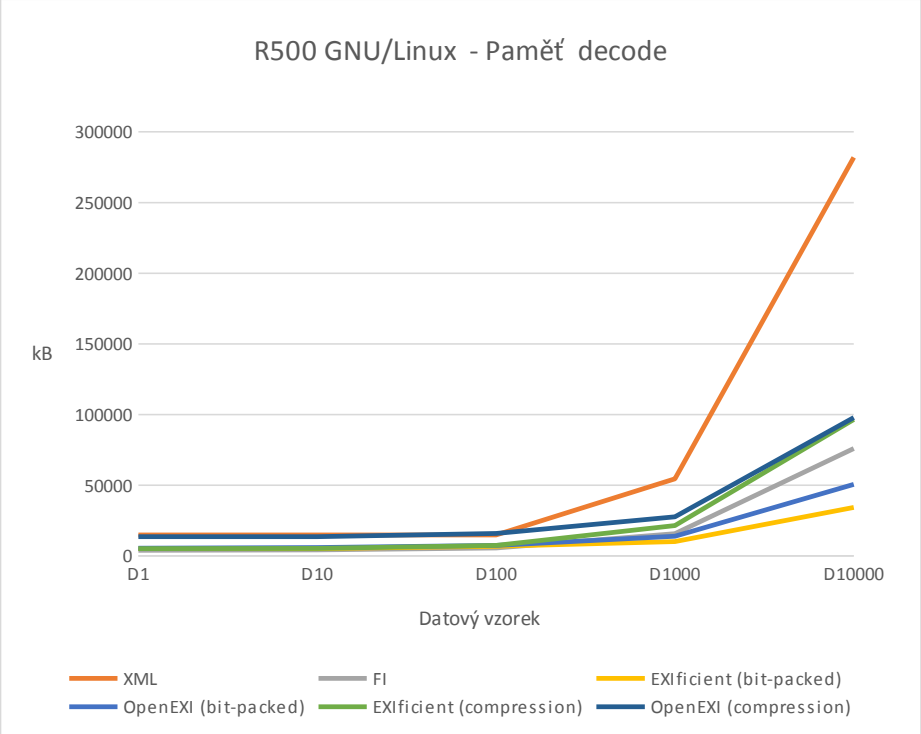
Obrázek 6.8: R500 - GNU/Linux paměťové nároky encode



Obrázek 6.9: R500 - GNU/Linux celkové paměťové nároky encode

	D1	D10	D100	D1000	D10000
XML	14821	14821	14821	54546	281922
FI	3886	4225	5637	15689	75984
EXIficient (bit-packed)	5131	5147	6706	10113	34260
OpenEXI (bit-packed)	5492	5825	7406	13862	50485
EXIficient (compression)	5161	5218	7399	21447	96498
OpenEXI (compression)	13609	13629	15872	27629	97934

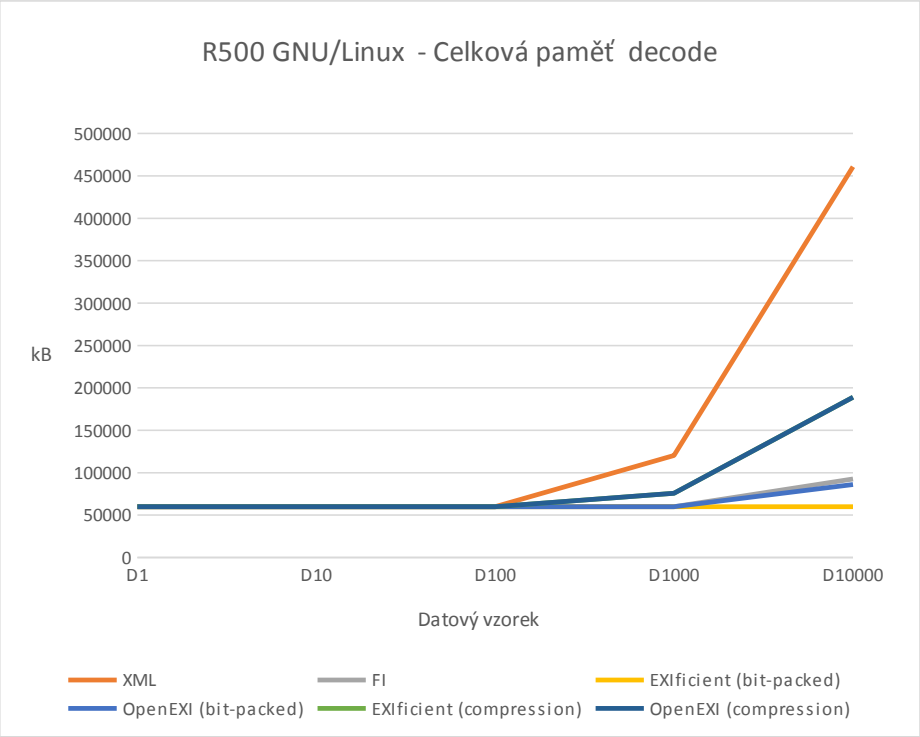
Tabulka 6.26: R500 - GNU/Linux paměťové nároky decode (kB)



Obrázek 6.10: R500 - GNU/Linux paměťové nároky decode

	D1	D10	D100	D1000	D10000
XML	59904	59904	59904	120320	460800
FI	59904	59904	59904	59904	92672
EXIficient (bit-packed)	59904	59904	59904	59904	59904
OpenEXI (bit-packed)	59904	59904	59904	59904	86016
EXIficient (compression)	59904	59904	59904	75776	188928
OpenEXI (compression)	59904	59904	59904	75776	188928

Tabulka 6.27: R500 - GNU/Linux celkové paměťové nároky decode (kB)

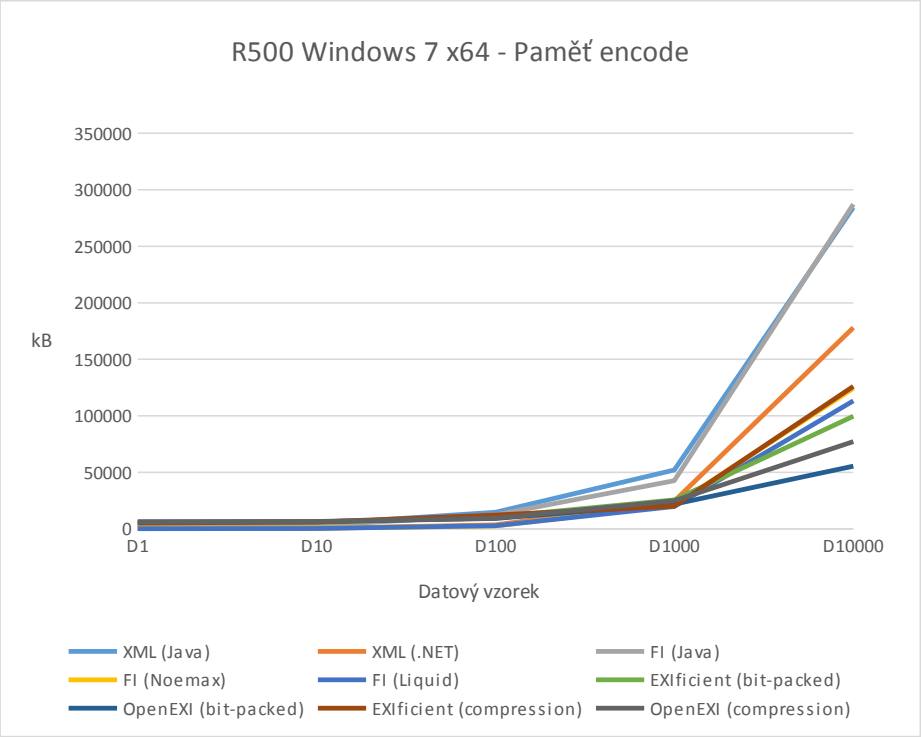


Obrázek 6.11: R500 - GNU/Linux celkové paměťové nároky decode

6.3.2 R500 - MS Windows 7 paměťové nároky

	D1	D10	D100	D1000	D10000
XML (Java)	3368	4353	14620	52125	284352
XML (.NET)	214	462	3561	24563	177920
FI (Java)	1706	2691	12415	42578	287126
FI (Noemax)	278	430	2632	20937	124479
FI (Liquid)	165	309	2785	19736	113324
EXIficient (bit-packed)	5311	5625	10521	25722	99592
OpenEXI (bit-packed)	6106	6277	9103	21938	55529
EXIficient (compression)	5328	5946	12124	20188	126133
OpenEXI (compression)	6245	6607	9304	24937	77269

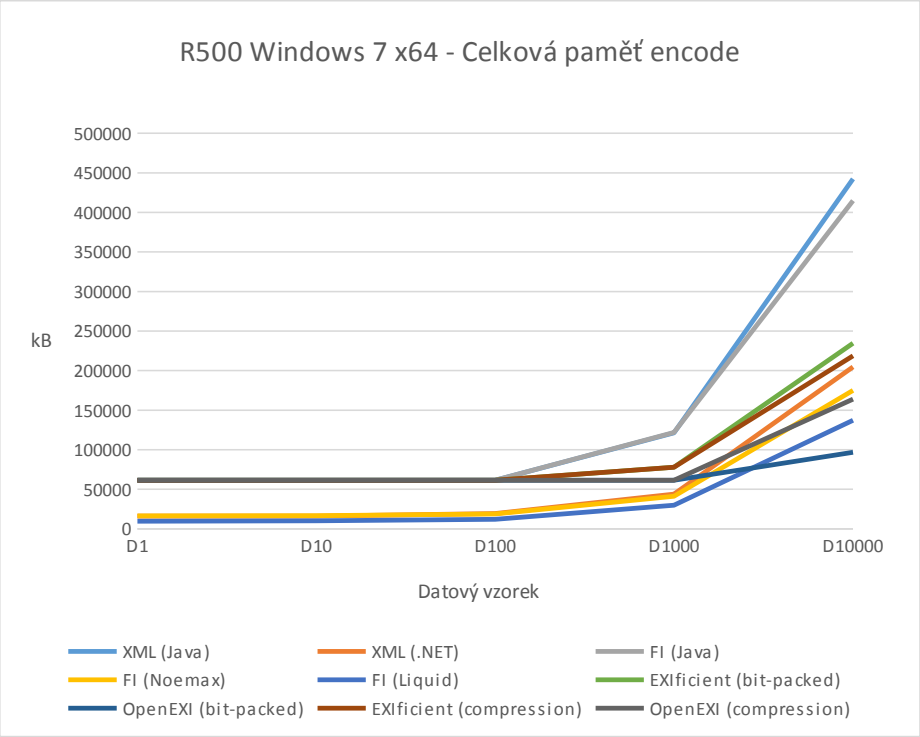
Tabulka 6.28: R500 - MS Windows 7 paměťové nároky encode (kB)



Obrázek 6.12: R500 - MS Windows 7 paměťové nároky encode

	D1	D10	D100	D1000	D10000
XML (Java)	61440	61440	61440	121344	442368
XML (.NET)	16136	16204	19304	43852	204712
FI (Java)	61440	61440	61440	121856	414720
FI (Noemax)	16304	16500	18884	41212	174952
FI (Liquid)	9596	10196	12208	29892	137152
EXIficient (bit-packed)	61440	61440	61440	77824	234568
OpenEXI (bit-packed)	61440	61440	61440	61440	96768
EXIficient (compression)	61440	61440	61440	77824	218624
OpenEXI (compression)	61440	61440	61440	61440	163840

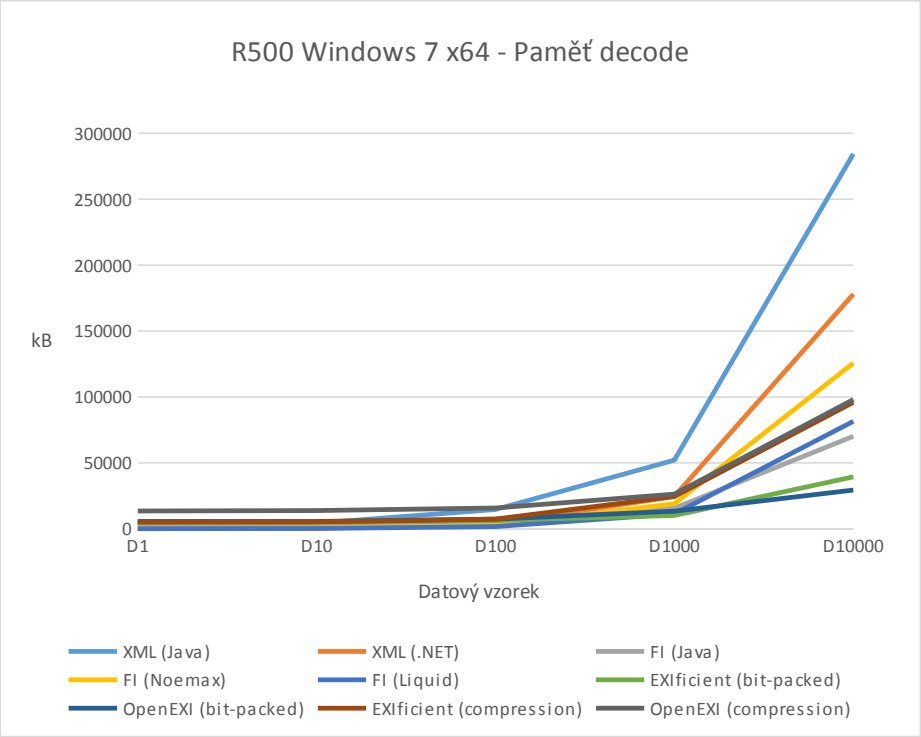
Tabulka 6.29: R500 - MS Windows 7 celkové paměťové nároky encode (kB)



Obrázek 6.13: R500 - MS Windows 7 celkové paměťové nároky encode

	D1	D10	D100	D1000	D10000
XML (Java)	3368	4353	14620	52125	284352
XML (.NET)	214	462	3561	24563	177920
FI (Java)	3950	4034	5389	15668	70249
FI (Noemax)	286	526	2496	19039	125637
FI (Liquid)	165	341	1773	11268	81477
EXIficient (bit-packed)	5055	4993	6448	10146	39536
OpenEXI (bit-packed)	5585	5603	7365	13292	29431
EXIficient (compression)	5108	5334	7332	24524	95935
OpenEXI (compression)	13480	13821	15762	26331	98069

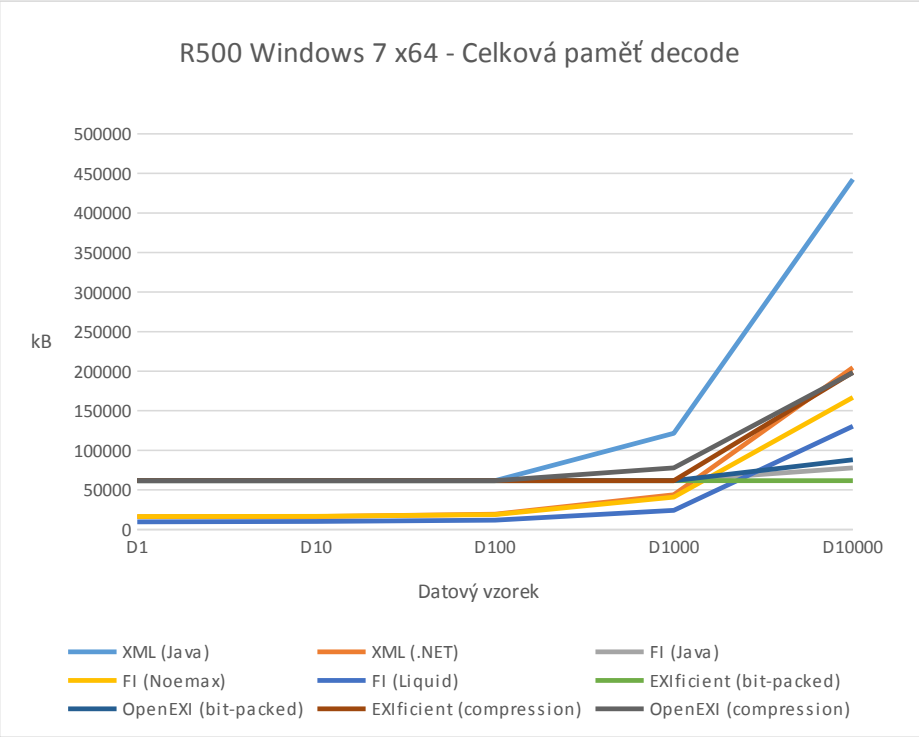
Tabulka 6.30: R500 - MS Windows 7 paměťové nároky decode (kB)



Obrázek 6.14: R500 - MS Windows 7 paměťové nároky decode

	D1	D10	D100	D1000	D10000
XML (Java)	61440	61440	61440	121344	442368
XML (.NET)	16136	16204	19304	43852	204712
FI (Java)	61440	61440	61440	61440	77824
FI (Noemax)	16300	16568	18780	40872	166944
FI (Liquid)	9576	10204	11612	24160	130452
EXIficient (bit-packed)	61440	61440	61440	61440	61440
OpenEXI (bit-packed)	61440	61440	61440	61440	88064
EXIficient (compression)	61440	61440	61440	61440	198656
OpenEXI (compression)	61440	61440	61440	77824	198144

Tabulka 6.31: R500 - MS Windows celkové paměťové nároky decode (kB)



Obrázek 6.15: R500 - MS Windows 7 celkové paměťové nároky decode

6.3.3 Srovnání paměťových nároků

Srovnání paměťových nároků mezi operačními systémy GNU/Linux a MS Windows je poměrně vyrovnané, protože dostupné implementace pro oba systémy jsou napsány v jazyce Java, které využívá již dříve diskutovanou pre-alokaci paměti pro proces. Hodnota takto vyhrazené paměti je dostatečná pro většinu testů, a proto jsou výsledky velice podobné. Díky této vlastnosti vychází ze srovnání lépe GNU/Linux, protože využívá v o 2.5% menší množství pre-alokované paměti.

Rozdíly ve výkonu jsou v následujících tabulkách vypočítány následovně:

- **Rozdíl** (celková paměť v Linuxu) - (celková paměť ve Windows)
- **Rozdíl v %** 1 - (celková paměť ve Windows) / (celková paměť v Linuxu)

	D1	**D10**	**D100**	**D1000**	**D10000**
XML (Java, Linux)	59904	59904	59904	120320	460800
XML (Java)	61440	61440	61440	121344	442368
Rozdíl	-1536	-1536	-1536	-1024	18432
Rozdíl v %	2.50%	2.50%	2.50%	0.84%	-4.17%

Tabulka 6.32: Porovnání výkonu nativních parserů (kB)

	D1	D10	D100	D1000	D10000
FI (Java, Linux)	59904	59904	59904	118784	437248
FI (Java, Windows)	61440	61440	61440	121856	414720
Rozdíl	-1536	-1536	-1536	-3072	22528
Rozdíl v %	2.50%	2.50%	2.50%	2.52%	-5.43%

Tabulka 6.33: Porovnání výkonu nativní implementace Fast Infosetu, encode (kB)

	D1	D10	D100	D1000	D10000
EXIficient (bit-packed, Linux)	59904	59904	59904	75776	166400
EXIficient (bit-packed, Windows)	61440	61440	61440	77824	234568
Rozdíl	-1536	-1536	-1536	-2048	-68168
Rozdíl v %	2.50%	2.50%	2.50%	2.63%	29.06%

Tabulka 6.34: Porovnání výkonu EXIficient, bit-packed, encode (kB)

	D1	D10	D100	D1000	D10000
OpenEXI (bit-packed, Linux)	59904	59904	59904	59904	94720
OpenEXI (bit-packed, Windows)	61440	61440	61440	61440	96768
Rozdíl	-1536	-1536	-1536	-1536	-2048
Rozdíl v %	2.50%	2.50%	2.50%	2.50%	2.12%

Tabulka 6.35: Porovnání výkonu OpenEXI, bit-packed, encode (kB)

	D1	D10	D100	D1000	D10000
EXIficient (compression, Linux)	59904	59904	59904	75776	219136
EXIficient (compression, Windows)	61440	61440	61440	77824	218624
Rozdíl	-1536	-1536	-1536	-2048	512
Rozdíl v %	2.50%	2.50%	2.50%	2.63%	-0.23%

Tabulka 6.36: Porovnání výkonu EXIficient, compression, encode (kB)

	D1	D10	D100	D1000	D10000
OpenEXI (compression, Linux)	59904	59904	59904	75776	157696
OpenEXI (compression, Windows)	61440	61440	61440	61440	163840
Rozdíl	-1536	-1536	-1536	14336	-6144
Rozdíl v %	2.50%	2.50%	2.50%	-23.33%	3.75%

Tabulka 6.37: Porovnání výkonu OpenEXI, compression, encode (kB)

	D1	D10	D100	D1000	D10000
FI (Java, Linux)	59904	59904	59904	59904	92672
FI (Java, Windows)	61440	61440	61440	61440	77824
Rozdíl	-1536	-1536	-1536	-1536	14848
Rozdíl v %	2.50%	2.50%	2.50%	2.50%	-19.08%

Tabulka 6.38: Porovnání výkonu nativní implementace Fast Infosetu, decode (kB)

	D1	**D10**	**D100**	**D1000**	**D10000**
EXIficient (bit-packed, Linux)	59904	59904	59904	59904	59904
EXIficient (bit-packed, Windows)	61440	61440	61440	61440	61440
Rozdíl	-1536	-1536	-1536	-1536	-1536
Rozdíl v %	2.50%	2.50%	2.50%	2.50%	2.50%

Tabulka 6.39: Porovnání výkonu EXIficient, bit-packed, decode (kB)

Knihovna/Pocet instanci	**D1**	**D10**	**D100**	**D1000**	**D10000**
OpenEXI (bit-packed, Linux)	59904	59904	59904	59904	86016
OpenEXI (bit-packed, Windows)	61440	61440	61440	61440	88064
Rozdíl	-1536	-1536	-1536	-1536	-2048
Rozdíl v %	2.50%	2.50%	2.50%	2.50%	2.33%

Tabulka 6.40: Porovnání výkonu OpenEXI, bit-packed, decode (kB)

	D1	**D10**	**D100**	**D1000**	**D10000**
EXIficient (compression, Linux)	59904	59904	59904	75776	188928
EXIficient (compression, Windows)	61440	61440	61440	61440	198656
Rozdíl	-1536	-1536	-1536	14336	-9728
Rozdíl v %	2.50%	2.50%	2.50%	-23.33%	4.90%

Tabulka 6.41: Porovnání výkonu EXIficient, compression, decode (kB)

Knihovna/Pocet instanci	**D1**	**D10**	**D100**	**D1000**	**D10000**
OpenEXI (compression, Linux)	59904	59904	59904	75776	188928
OpenEXI (compression, Windows)	61440	61440	61440	77824	198144
Rozdíl	-1536	-1536	-1536	-2048	-9216
Rozdíl v %	2.50%	2.50%	2.50%	2.63%	4.65%

Tabulka 6.42: Porovnání výkonu OpenEXI, compression, decode (kB)

6.4 Vliv použití jmenných prostorů

Při použití jmenných prostorů byl zaznamenán měřitelný propad výkonu u aplikací na platformě Java, kde byl výkonový dopad u větších instancí řádově 15%, na platformě .NET nebyl dopad tak výrazný a v některých případech byly naměřeny menší hodnoty než v případě testovacích souborů bez jmenných prostorů. Tento test byl prováděn pouze na konfiguraci ProBook 6560 a byl zaměřen pouze na rychlost konverze[7]. Velikost výsledných souborů byla ovlivněna více u Fast Infosetu, kde došlo k navýšení velikosti u testovacího souboru D10000 na 2810 kB, což je nárůst o 135 kB (4.8%). Nárůst velikosti souboru u EXI byl řádu několika kB u souboru D10000, při použití komprese byl ještě menší.

	D1	**D10**	**D100**	**D1000**	**D10000**
XML (Java)	0.68	1.91	12.99	113.37	1035.97
XML s NS (Java)	0.67	2.02	13.77	120.50	1231.40
XML (.NET)	0.03	0.27	3.18	27.57	225.64
XML s NS (.NET)	0.04	0.28	3.27	31.47	231.46
FI (Java)	0.95	2.15	13.45	120.58	1181.24
FI s NS (Java)	0.80	2.11	15.95	142.88	1316.30
FI (Noemax)	0.03	0.22	2.66	30.25	201.25
FI s NS (Noemax)	0.04	0.26	2.56	30.60	192.39
FI (Liquid)	0.03	0.23	2.07	24.21	178.13
FI s NS (Liquid)	0.03	0.24	2.18	25.21	187.75
EXIficient (bit-packed)	0.50	1.35	9.02	79.80	536.41
EXIficient s NS (bit-packed)	0.85	1.51	9.79	91.93	613.74
OpenEXI (bit-packed)	0.83	1.69	9.56	73.05	502.24
OpenEXI s NS (bit-packed)	1.03	2.26	11.36	89.52	570.04
EXIficient (compression)	1.21	4.25	32.32	289.81	1889.48
EXIficient s NS (compression)	1.09	4.54	35.30	307.64	1961.81
OpenEXI (compression)	1.13	2.01	11.09	118.37	655.37
OpenEXI s NS (compression)	1.03	5.48	13.15	141.81	794.78

Tabulka 6.43: Porovnání výkonu při encode a použití jmenných prostorů a bez nich (ms)

[7]Při testování paměti byly nalezeny odpovídající nárůsty v použité paměti, ale tento rozdíl se zejména v případě Javy, u menších instancí ztratil ve výchozím množství alokované paměti a výsledky testů by tedy brzy vypadaly jako v případě testu bez jmenných prostorů.

	D1	D10	D100	D1000	D10000
FI (Java)	0.24	0.59	3.72	31.55	234.70
FI s NS (Java)	0.27	0.73	4.87	41.14	331.92
FI (Noemax)	0.03	0.35	2.24	28.04	218.03
FI s NS (Noemax)	0.04	0.23	2.17	27.23	234.17
FI (Liquid)	0.02	0.17	1.77	22.10	167.32
FI s NS (Liquid)	0.02	0.45	1.88	21.43	178.99
EXIficient (bit-packed)	0.36	0.94	5.68	45.76	311.06
EXIficient s NS (bit-packed)	0.45	1.21	6.99	59.40	386.48
OpenEXI (bit-packed)	0.51	1.19	7.89	47.16	327.78
OpenEXI s NS (bit-packed)	1.09	1.52	6.00	56.28	376.03
EXIficient (compression)	0.65	2.79	20.77	181.75	1194.20
EXIficient s NS (compression)	0.77	2.90	22.56	199.97	1317.59
OpenEXI (compression)	1.34	2.05	6.14	47.92	402.50
OpenEXI s NS (compression)	1.41	2.75	7.96	58.47	461.08

Tabulka 6.44: Porovnání výkonu při decode a použití jmenných prostorů a bez nich (ms)

6.5 Vliv XML Schema na výkon EXI

Použití standardu EXI s přiloženou definicí XML Schema přináší pozitivní výsledky. Při použití XML Schema a zarovnání souboru bit-packed dochází ke zmenšení výsledného souboru o cca 10% při zachování srovnatelného (nebo mírně vyššího výkonu) při konverzi oběma směry, při použití varianty s kompresí výstupního souboru je zachována mírná výkonnostní výhoda, ale změna ve velikosti výstupního souboru je zanedbatelná a v případě testovacího vzorku D10000 dochází k jeho zvětšení oproti variantě bez XML Schema.

Další výhodou, kterou nám použití XML Schema nabízí (nejen v případě EXI) je validace výstupních dat. EXI podporuje i striktní režim interpretace XML Schema, kdy jakákoli odchylka od definovaného Schema zajistí vyvolání výjimky a ukončení konverze.

Při testování nebyl do času na zpracování zahrnut čas nutný na parsování a přípravu EXI Schema ze souboru s XML Schematem. Toto rozhodnutí bylo založeno na předpokládaném způsobu použití - EXI Schema je struktura s dlouhou životností, podobně jako XML Schema je poměrně stálá a není tedy nutné vytvářet její novou reprezentaci pro každou konverzi. Také můžeme bez větších potíží předpokládat, že tato struktura bude držena v operační paměti serveru pro rychlé použití. Toto rozhodnutí se projevilo zvýšenou spotřebou operační paměti.

	D1	D10	D100	D1000	D10000
XML	4.1	37.5	369.9	3585.4	22603.9
XML (lineární)	2.8	25.9	254.7	2467.8	15717.2
OpenEXI (bit-packed)	1.1	4.8	35.8	282.3	1721.5
OpenEXI (bit-packed, XSD)	0.4	3.6	28.7	253.5	1589.5
OpenEXI (compression)	0.7	2.5	15.5	107.6	616.9
OpenEXI (compression, XSD)	0.4	2.0	14.7	106.3	626.0
EXIficient (bit-packed)	1.1	4.8	35.8	282.3	1721.5
EXIficient (bit-packed, XSD)	0.4	3.6	28.9	255.6	1602.2
EXIficient(compression)	0.7	2.5	15.5	107.6	616.9
EXIficient(compression, XSD)	0.4	2.0	14.6	106.2	625.8

Tabulka 6.45: Porovnání velikostí výstupních souborů při použití XML Schema a bez něj (kB)

	D1	D10	D100	D1000	D10000
EXIficient (bit-packed)	0.50	1.35	9.02	79.80	536.41
EXIficient (bit-packed, XSD)	0.43	1.13	8.33	77.57	480.13
OpenEXI (bit-packed)	0.83	1.69	9.56	73.05	502.24
OpenEXI (bit-packed, XSD)	0.81	1.73	9.32	73.96	459.51
EXIficient (compression)	1.21	4.25	32.32	289.81	1889.48
EXIficient (compression, XSD)	0.79	4.82	34.96	313.64	2031.57
OpenEXI (compression)	1.13	2.01	11.09	118.37	655.37
OpenEXI (compression, XSD)	0.91	2.17	12.75	101.77	676.07

Tabulka 6.46: ProBook 6560 - MS Windows 7 čas encode s XML Schema (ms)

	D1	D10	D100	D1000	D10000
EXIficient (bit-packed)	5350	6024	11253	27255	149800
EXIficient (bit-packed, XSD)	10737	11393	16571	35919	132227
OpenEXI (bit-packed)	6050	6667	9347	29529	125279
OpenEXI (bit-packed, XSD)	11524	12117	15002	23816	93785
EXIficient (compression)	5350	6017	12611	17652	151615
EXIficient (compression, XSD)	10737	11403	19872	38079	126694
OpenEXI (compression)	6728	6731	10103	32434	76833
OpenEXI (compression, XSD)	11524	12192	15684	19971	89416

Tabulka 6.47: ProBook 6560 - MS Windows 7 paměť encode s XML Schema (kB)

	D1	D10	D100	D1000	D10000
EXIficient (bit-packed)	125440	125440	125440	125440	225280
EXIficient (bit-packed, XSD)	125440	125440	125440	125440	158720
OpenEXI (bit-packed)	125440	125440	125440	125440	158720
OpenEXI (bit-packed, XSD)	125440	125440	125440	125440	157696
EXIficient (compression)	125440	125440	125440	125440	247296
EXIficient (compression, XSD)	125440	125440	125440	125440	265728
OpenEXI (compression)	125440	125440	125440	125440	239616
OpenEXI (compression, XSD)	125440	125440	125440	125440	250880

Tabulka 6.48: ProBook 6560 - MS Windows 7 celková paměť encode s XML Schema (kB)

	D1	D10	D100	D1000	D10000
EXIficient (bit-packed)	0.36	0.94	5.68	45.76	311.06
EXIficient (bit-packed, XSD)	0.62	0.93	5.64	47.65	302.49
OpenEXI (bit-packed)	0.51	1.19	7.89	47.16	327.78
OpenEXI (bit-packed, XSD)	0.54	1.16	9.59	47.58	302.83
EXIficient (compression)	0.65	2.79	20.77	181.75	1194.2
EXIficient (compression, XSD)	0.69	2.66	20.96	185.53	1327.65
OpenEXI (compression)	1.34	2.05	6.14	47.92	402.5
OpenEXI (compression, XSD)	1.27	1.85	6.73	49.49	351.18

Tabulka 6.49: ProBook 6560 - MS Windows 7 čas decode s XML Schema (ms)

	D1	D10	D100	D1000	D10000
EXIficient (bit-packed)	5395	5397	6819	20361	44533
EXIficient (bit-packed, XSD)	12071	12107	14148	30315	40088
OpenEXI (bit-packed)	6004	6020	7539	24338	62900
OpenEXI (bit-packed, XSD)	12771	12794	14445	30108	40158
EXIficient (compression)	5402	5344	7430	32032	96201
EXIficient (compression, XSD)	12093	12763	15065	28880	128906
OpenEXI (compression)	13824	13859	15819	26299	97876
OpenEXI (compression, XSD)	20577	21258	23558	37203	133937

Tabulka 6.50: ProBook 6560 - MS Windows 7 paměť decode s XML Schema (kB)

	D1	D10	D100	D1000	D10000
EXIficient (bit-packed)	125440	125440	125440	125440	125440
EXIficient (bit-packed, XSD)	125440	125440	125440	125440	125440
OpenEXI (bit-packed)	125440	125440	125440	125440	158720
OpenEXI (bit-packed, XSD)	125440	125440	125440	125440	125440
EXIficient (compression)	125440	125440	125440	125440	264704
EXIficient (compression, XSD)	125440	125440	125440	125440	273920
OpenEXI (compression)	125440	125440	125440	125440	226304
OpenEXI (compression, XSD)	125440	125440	125440	125440	283648

Tabulka 6.51: ProBook 6560 - MS Windows 7 celková paměť decode s XML Schema (kB)

6.6 Interoperabilita implementací

Interoperabilita jednotlivých implementací byla testována křížovým zpracováním souborů vytvořených ostatními implementacemi a následným porovnáním jejich výstupů a zdrojového XML souboru pomocí utility Pretty Diff[8] ve webovém prohlížeči. Test byl prováděn s datovým vzorkem D1000 (ve variantě se jmennými prostory i bez nich). U EXI byly testovány kombinace zarovnání souboru bit-packed a compression v kombinaci s (ne)použitím XML Schema.

6.6.1 Fast Infoset

V implementacích Fast Infosetu nebyly nalezeny žádné rozdíly a vzájemná konverze souborů fungovala bez problémů, ale v implementacích z Javy a Liquid Fast Infoset byly nalezeny nedostatky. Implementace z Javy obsahuje chybu[9], která neodstraňuje korektně všechny "bílé"znaky a nedochází tedy k využití plného potenciálu algoritmu. Ukázka fragmentu souboru s "bílými"znaky:

```
customers?
  ????customer??
    ??? customerId??11000????forename??Mary????surname??Young?????
emailPromotion?0??????address??7246 Ptarmigan Drive??????city??Renton?????
```

Druhým problémem, který byl odhalen v implementaci Liquid Fast Infoset, je nesprávné ukončení souboru, které vyvolá výjimku při otevírání souboru - Fast Infoset parser ohlásí neočekávaný konec souboru. Hrubé řešení tohoto problému je připojení tří bytů `0xFF` na konec výstupního souboru. Po této drobné úpravě byly generované soubory bez problémů zpracovány v ostatních implementacích Fast Infosetu.

6.6.2 EXI

U implementací EXI nebyly nalezeny žádné problémy s kompatibilitou binárních verzí souborů, ale byly nalezeny dva rozdíly u souborů rekonstruovaných z binární podoby. Prvním drobným rozdílem je odlišná deklarace použití XML Schema na začátku konvertovaných souborů - viz příklad níže:

```
<customers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<customers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ns3="http://www.w3.org/2001/XMLSchema">
```

Druhým rozdílem, který může být v některých případech poněkud nepříjemný, je používání generovaných prefixů k jmenným prostorům a jejich lokální definice. V tomto případě nejen

[8] PrettyDiff je dostupný na adrese: <http://prettydiff.com/>
[9] Chyba byla zmiňována v diskuzi výsledků velikostí souborů.

že nezůstávají zachovány původní prefixy jmenných prostorů, ale obě implementace používají jiný algoritmus pro jejich generování. Při použití jmenných prostorů v EXI je jim nutné věnovat zvýšenou pozornost. Následující ukázky náleží těmto případům[10].

6.6.2.1 Původní soubor

```
<c:customers xmlns:c="urn:corp:customer">
  <c:customer>
    <c:customerId>11000</c:customerId>
    <c:forename>Mary</c:forename>
    <c:surname>Young</c:surname>
    <o:orders xmlns:o="urn:corp:order">
      <o:order>
        <o:orderId>43793</o:orderId>
        <o:salesOrderNumber>SO43793</o:salesOrderNumber>
```

6.6.2.2 OpenEXI

```
<p0:customers xmlns:p0="urn:corp:customer"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <p0:customer><p0:customerId>11000</p0:customerId>
        <p0:forename>Mary</p0:forename>
        <p0:surname>Young</p0:surname>
        <p1:orders xmlns:p1="urn:corp:order">
            <p1:order><p1:orderId>43793</p1:orderId>
                <p1:salesOrderNumber>SO43793</p1:salesOrderNumber>
```

6.6.2.3 EXIficient

```
<ns4:customers xmlns:ns3="http://www.w3.org/2001/XMLSchema"
xmlns:ns4="urn:corp:customer"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ns4:customer><ns4:customerId>11000</ns4:customerId>
        <ns4:forename>Mary</ns4:forename>
        <ns4:surname>Young</ns4:surname>
        <ns5:orders xmlns:ns5="urn:corp:order">
            <ns5:order><ns5:orderId>43793</ns5:orderId>
                <ns5:salesOrderNumber>SO43793</ns5:salesOrderNumber>
```

[10] Pro zvýšení přehlednosti byly ukázky zkráceny, vynechané řádky neměly vliv na diskutovanou skutečnost.

6.7 Shrnutí výsledků

V průběhu testování bylo ověřeno, že oba standardy, Fast Infoset i Efficient XML Interchange, splňují očekávání, která jsou na ně kladena. Svého primárního úkolu, efektivní reprezentace XML dat v binární formě a tím snížení nároků na velikost výsledného souboru a zároveň snížení výpočetní náročnosti, snadno dosahují. Porovnání velikostí obou souborů ukazuje na jasnou převahu EXI, které dosahuje[11] až 4x menší velikosti souborů než konkurenční Fast Infoset.

Při časovém srovnání už výsledky nejsou takto jednoznačné - pokud budeme uvažovat platformu Java, která se ukázala jako výrazně pomalejší než .NET, potom je Fast Infoset rychlejší při zpětné konverzi z binárního formátu do XML, ale EXI nabízí typicky vyšší rychlost zpracování XML do binární podoby. Důvodem pro rychlejší zpracování XML v EXI je pravděpodobně použití proudového zpracování místo budování DOM. Pro platformu .NET zatím není k dispozici Open Source implementace EXI a zkušební verzi komerční knihovny od firmy Agile Data Inc. se mi nepodařilo získat pro test[12], a proto byl testován pouze Fast Infoset, který (až na drobné problémy v implementaci Liquid Fast Infoset) byl rychlejší než parsování XML souboru i když nebyl tento výkonový odskok tak výrazný jako v případě Javy.

Spotřeba paměti je při zpracování podle očekávání vysoká. Z tohoto hlediska se jako nejúspornější ukázaly implementace v .NETu, které mají výrazně níže položenou minimální velikost alokované paměti, a proto výrazně lépe škálují velikost alokované paměti než implementace v Javě, které alokují nad základní velikost až u dat velikosti D1000 a větších.

Porovnání výkonu mezi operačními systémy GNU/Linux a MS Windows 7 a jejich implementací Javy nemá jednoznačného vítěze - OpenJDK na Linuxu používá rychlejší XML parser, ale ztrácí při zpětné konverzi binárních formátů do XML a naopak. Z hlediska využité paměti si jsou obě platformy rovnocenné.

Na následujících grafech je zobrazeno srovnání rychlosti zpracování vstupního souboru při encode/decode na ose X, množství spotřebované paměti na ose Y a velikost výstupního (při encode) nebo vstupního (při decode) souboru jako plocha kruhu. Srovnávaná data pochází z měření na ProBooku 6560 s daty D10000 (největší instance).

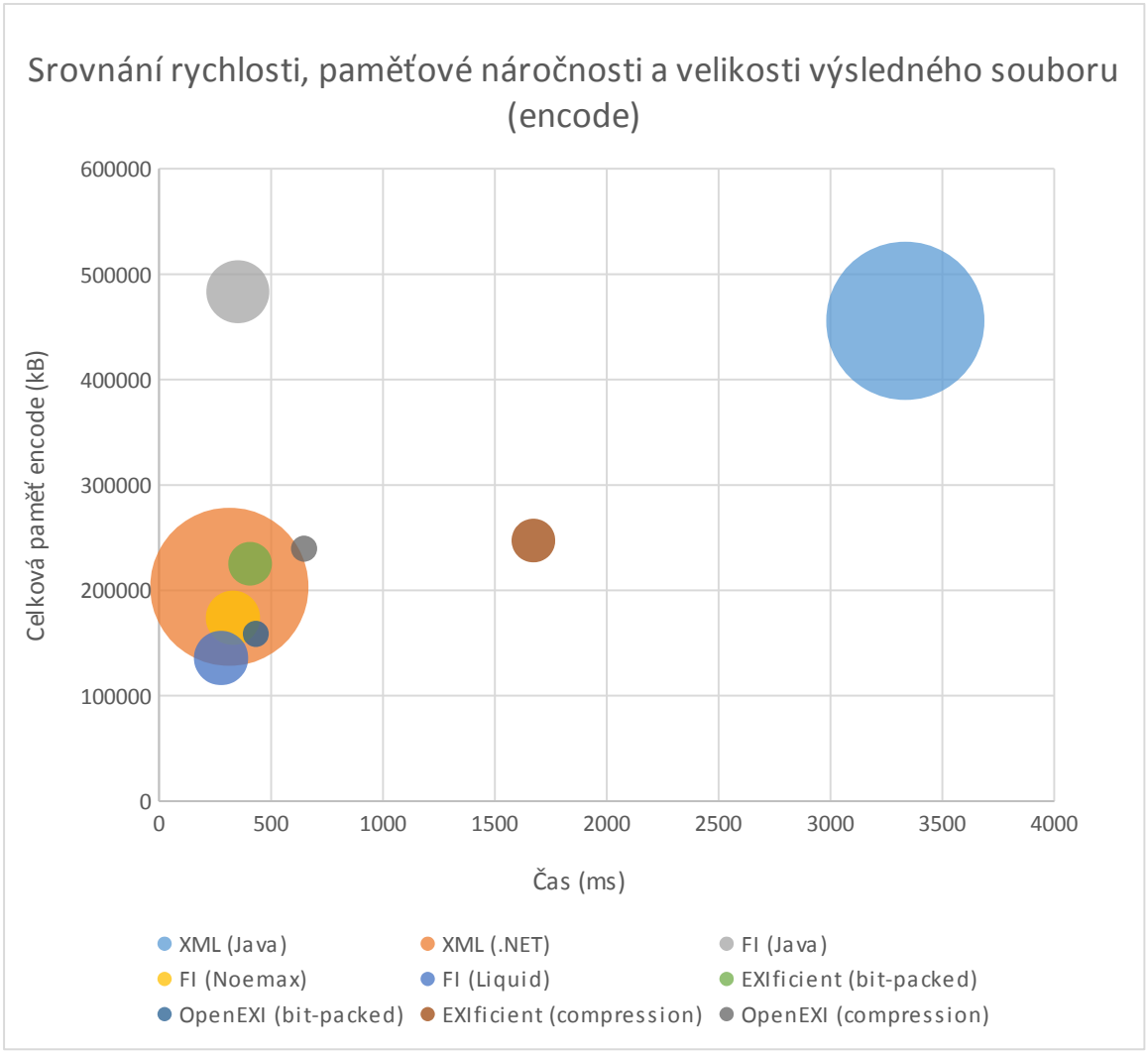
Z implementací jednotlivých standardů vychází pro Fast Infoset nejlépe Liquid Fast Infoset pro platformu .NET, který vykazuje stabilní výkon a nízkou spotřebu paměti (zejména v porovnání s implementací v Javě). Další výhodou této knihovny je i to, že je poskytována pod GNU Affero GPL licencí. Pro standard EXI dosahuje stabilně lepších výsledků implementace OpenEXI, která je méně náročná na operační paměť a podává stabilně dobrý výkon, obzvláště při použití komprese je výrazně výkonnější než konkurenční EXIficient. Obě testované knihovny pro EXI jsou poskytovány pod otevřenými licencemi Apache Licence/GNU GPL.

Zajímavé výsledky ukazuje i srovnání hrubého výkonu obou testovacích strojů, které od sebe dělí 2 generace procesorů (jedná se o procesory stejné kategorie a cenové relace v době uvedení na trh) a naměřený rozdíl mezi jejich výkonem se pohybuje okolo hodnoty 30%. Do

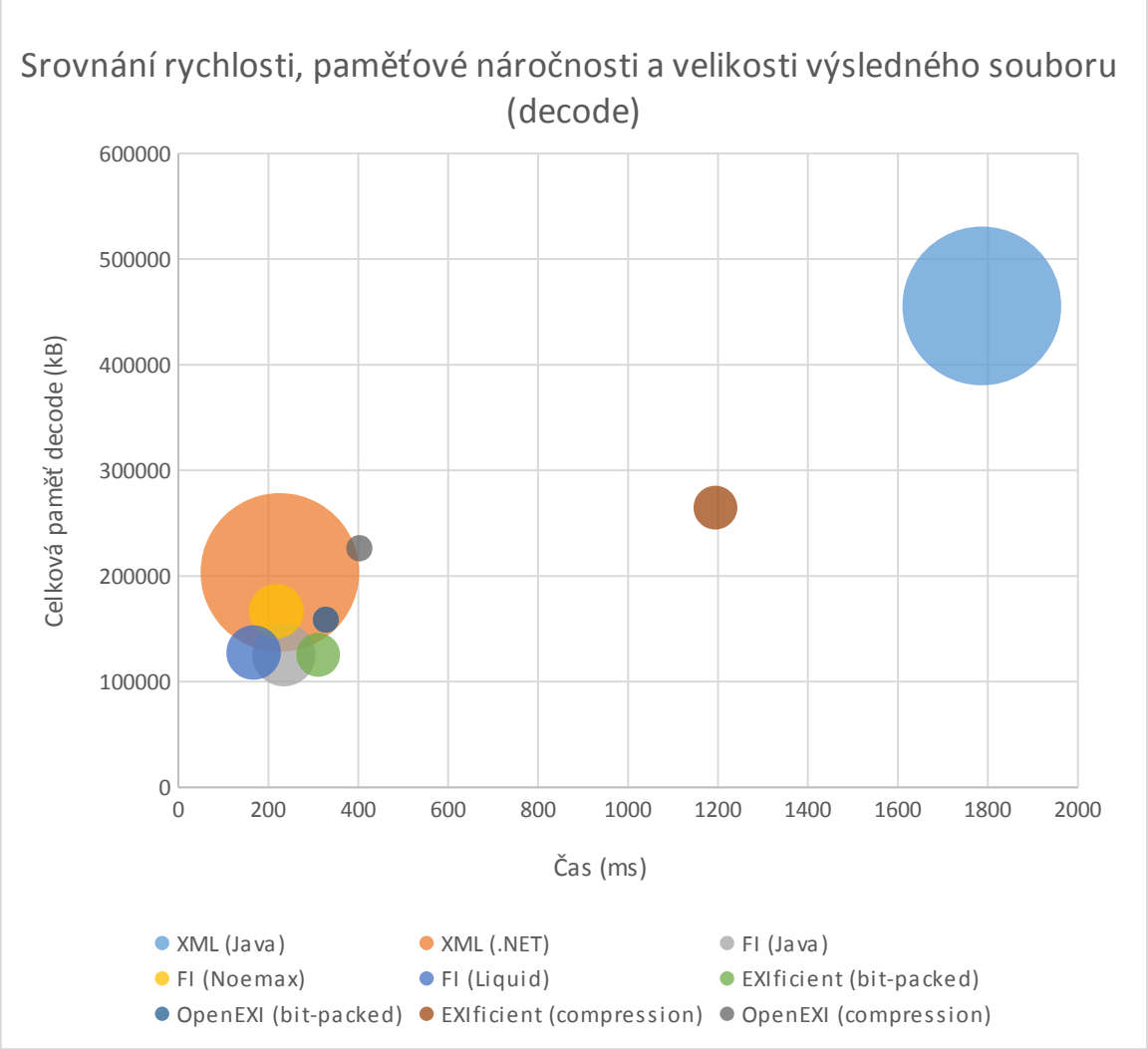
[11] Na testovacích datech.

[12] Tato zkušební verze není volně ke stažení - požadavek na poskytnutí testovací verze je schvalován pracovníky Agile Data Inc. a nebyl vyřízen do doby testování.

tohoto srovnání se může do jisté míry promítnout i dvojnásobné množství operační paměti v ProBooku 6560 a její vyšší frekvence.



Obrázek 6.16: Srovnání rychlosti, paměťové náročnosti a velikosti výsledného souboru - encode



Obrázek 6.17: Srovnání rychlosti, paměťové náročnosti a velikosti výsledného souboru - decode

Kapitola 7

Integrace do existujících systémů

Samotná existence standardů Fast Infoset a jejich implementací by nebyla mnoho platná, pokud by nebylo možné je integrovat do současných systémů a/nebo frameworků. Možnost integrace jednotlivých implementací Fast Infosetu a EXI do systémů využívajících XML jako výstupní formát souborů je poměrně přímočará - po přidání příslušné knihovny do projektu je tato knihovna využita k serializaci obsahu přímo do souboru, v případě načítání stačí využít parser obsažený v knihovně využívaného projektu. Tímto způsobem můžeme velice snadno začít využívat výhod binárních XML souborů, které byly podrobněji popsány v předchozích kapitolách[1].

Primárním uplatněním Fast Infosetu a EXI je využití těchto standardů jako efektivního kódování přenášených dat ve webových službách. Pro otestování možností integrace těchto "nových" formátů do frameworlů pro tvorbu webových služeb byly (po dohodě s vedoucím práce) vybrány dva populární frameworky:

- **JAX-WS**[2] pro Javu
- **Windows Communication Foundation (WCF)** pro platformu Microsoft .NET a webový server Internet Information Services (IIS)

7.1 Fast Infoset

7.1.1 JAX-WS

Použití Fast Infosetu v rámci webové služby, která je založena na JAX-WS je velice jednoduché. Vzhledem k tomu, že implementace JAX-WS Metro obsahuje implementaci Fast Infosetu (tato implementace byla testována v předchozí kapitole), je podpora Fast Infosetu dostupná "out-of-the-box" - na straně serveru není vyžadována žádná konfigurace. Používané

[1] V závislosti na použité platformě a dalších aspektech projektu (licenční podmínky a jiné) může být nutné využít některou z dalších implementaci, která nebyla zmiňována v této práci.

[2] JAX-WS je pouze definice standardu, který musejí jednotlivé implementace dodržet, referenční implementací tohoto projektu je již dříve zmiňované Metro vyvíjené v rámci komunitního aplikačního serveru GlassFish. Více informací o tomto projektu a soubory ke stažení jsou dostupné na adrese: <https://jax-ws.java.net/>

kódování se zcela řídí na straně klienta, který může serveru oznámit požadované kódování zpráv pomocí protokolu pro Content Negotiation, který využívá HTTP hlavičky `Accept` a `Content-Type`. V klientovi tedy musíme po přidání reference na webovou službu a přidání příslušných knihoven pro Fast Infoset[3] přidat následující řádky, které zajistí pokus vyjednání kódování Fast Infoset:

```
MyService service = new MyService(new URL(...));
WsPort port = service.getWsPort();
Map<String, Object> ctxt = ((BindingProvider)port).getRequestContext();
ctxt.put("com.sun.xml.ws.client.ContentNegotiation", "pessimistic");
```

V ukázce kódu je použita pesimistická metoda pro určení kódování pro přenos, tedy první požadavek je odeslán ve formátu XML a pokud server odpoví ve Fast Infosetu, je toto kódování používáno po celou dobu existence objektu `port`. Druhou variantou je použití parametru `"optimistic"`, který využije Fast Infoset již pro první požadavek na webovou službu.

7.1.2 Windows Communication Foundation

Použití Fast Infosetu ve WCF je složitější než v JAX-WS, protože ve WCF neexistuje podpora pro toto kódování. Pro otestování jsem tedy zvolil knihovnu Liquid Fast Infoset[4], která zajistí vlastní kódování zpráv.

Rozšiřitelnost o další kódování je řešena pomocí komponenty Encoder, která je umístěna mimo hlavní průběh zprávy stackem (viz obrázek 7.1). Tato komponenta zajišťuje "překlad" z libovolného formátu, který je použit při přenosu, do formátu XML SOAP zprávy. Vlastní Encoder je navíc nezávislý na konkrétní službě - připojením[5] nového Encoderu vzniká ve službě nový endpoint (podobně jako při konfiguraci jiného standardního endpointu). Pro zprovoznění vlastního kódování budeme muset implementovat následující 3 hlavní třídy a změnit konfiguraci na endpointu, na kterém budeme provozovat toto kódování.

- **FastInfosetEncoder** je vlatní třída encoderu, která dědí od `MessageEncoder`. V této třídě je nutné implementovat metody `ReadMessage()` a `WriteMessage()`, které zajišťují převod původní zprávy do binárního formátu a bude následně odeslán klientovi. Obě výše zmiňované metody musí být označeny jako `override`.
- **FastInfosetMessageEncoderFactory** je používána k vytváření instancí `FastInfosetEncoder`, jedná se o použití návrhového vzoru Factory.
- **FastInfosetBindingElement** zajišťuje vytvoření propojovacího elementu, který zajistí možnost konfigurace použitého encoderu v konfiguračním souboru[6]. V této třídě

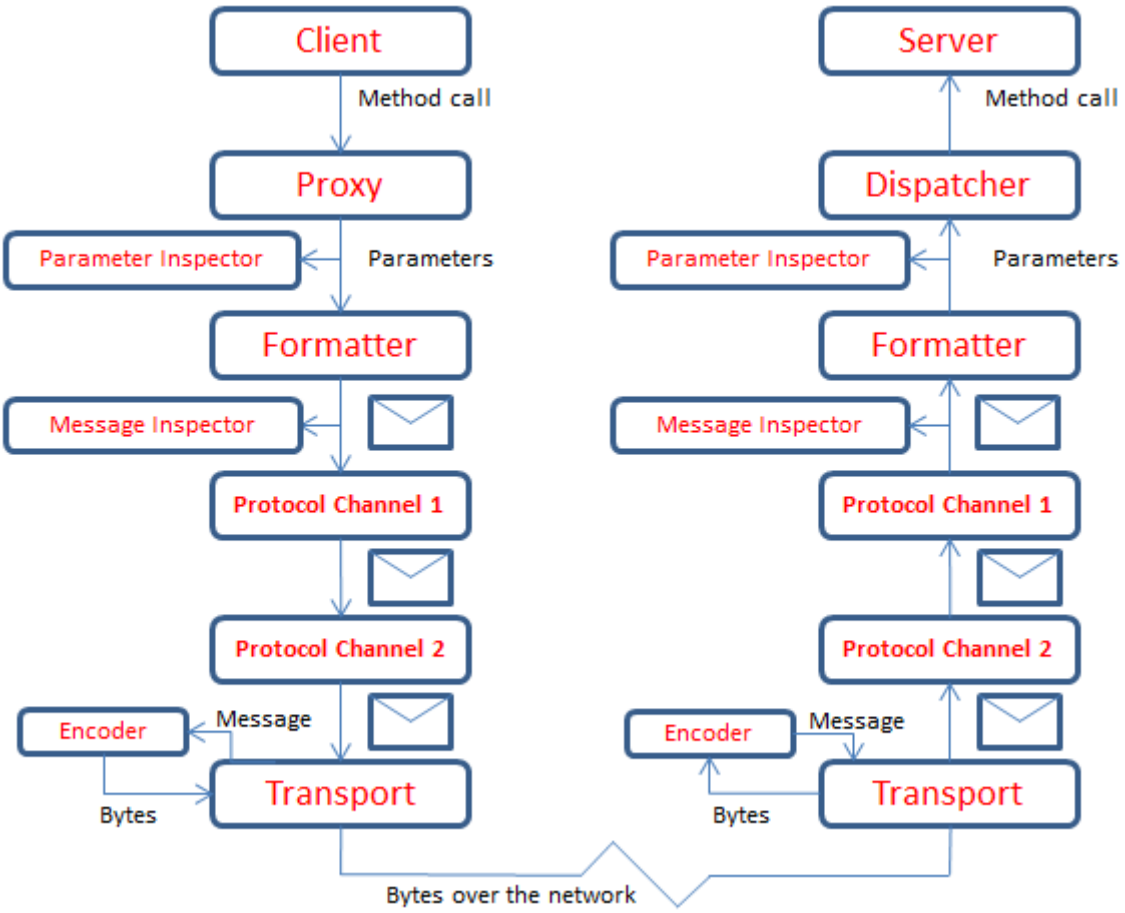
[3] Opět třeba z projektu Metro.

[4] Hlavním důvodem pro tuto volbu byly licenční podmínky (GNU Affero GPL), použití například FastInfoset.NET by bylo vzhledem k velice podobnému rozhraní velmi podobné.

[5] Konfigurací v souboru Web.config.

[6] Web.config v případě hostování webové služby na IIS serveru nebo App.config, pokud bude služba hostována například v konzolové aplikaci.

jsou volána dvě další podpůrné třídy `FastInfosetMessageEncodingElement` a `FastInfosetMessageEncodingBindingElementImporter` použití správného kódování uvnitř WCF stacku a zaregistrování nového enkodéru do WCF stacku.



Obrázek 7.1: Struktura zásobníku ve WCF[7]

Vlastní implementace webové služby se žádným způsobem neliší od typické WCF služby. Implementovaná služba obsahuje objektový model pro data používaná v průběhu testování výkonu, která jsou nejprve načtena do objektové podoby v paměti. Klient při volání webové služby požádá o počet instancí, které chce získat a webová služba vygeneruje odpověď, která tyto instance obsahuje.

Poslední věcí, kterou je nutné udělat na straně serveru je konfigurace. V konfiguračním souboru vytvoříme nový element `bindingElementExtension`, kterým se odkážeme na dříve vytvořený Encoder[8] a dále nakonfigurujeme (vytvoříme) `customBinding` element, který bude připojen k endpointu, na kterém bude webová služba naslouchat.

[7] Zdroj: http://blogs.msdn.com/cfs-filesystemfile.ashx/__key/communityserver-blogs-components-weblogfiles/00-00-00-95-96-metablogapi/1348.Channels_5F00_2E5A7836.png

[8] Zde je nutné si uvědomit, že ve vlastní webové službě nebyly provedeny žádné změny - encoder je zcela nezávislý na její implementaci a až v tuto chvíli je definováno jeho zapojení do WCF stacku.

```
<extensions>
  <bindingElementExtensions>
    <add name="fastInfosetMessageEncoding"
     type="FastInfosetEncoder.FastInfosetMessageEncodingElement,
     FastInfosetEncoder, Version=4.0.0.0, Culture=neutral, PublicKeyTo-
     ken=null" />
  </bindingElementExtensions>
</extensions>
<protocolMapping>
  <add scheme="http" binding="customBinding" />
</protocolMapping>
<bindings>
  <customBinding>
    <binding name="fastInfoset">
      <fastInfosetMessageEncoding innerMessageEncoding="textMessage-
      Encoding"/>
      <httpTransport hostNameComparisonMode="StrongWildcard"
       maxReceivedMessageSize="2147483647" />
    </binding>
  </customBinding>
</bindings>
```

Vlastní konfigurace endpointu je již standardní, použijeme dříve definovaný `fastInfoset` binding element a přiřadíme mu službu, kterou bude obsluhovat.

```
<services>
      <service name="WcfCustomerDataProvider.CustomerDataProvider"
       behaviorConfiguration="mexBehaviour">
        <endpoint address="CustomerDataProvider" binding="customBinding"
         bindingConfiguration="fastInfoset"
          contract="WcfCustomerDataProvider.ICustomerDataProvider">
        </endpoint>
        <endpoint address="mex" binding="mexHttpBinding" contract="IMeta-
        dataExchange" />
        <host>
          <baseAddresses>
            <add baseAddress="http://dev-server-web:8024/" />
          </baseAddresses>
        </host>
      </service>
    </services>
```

Na straně klienta je situace velice podobná, veškeré změny jsou prováděny v jeho konfiguraci (klient potřebuje, podobně jako server, mít k dispozici zkompilovanou knihovnu s Encoderem a všemi jeho závistlostmi).

Integrace nových/vlastních formátů do WCF je poměrně jednoduchá, jediným (očekávaným) problémem je na první pohled nepřehledná struktura implementovaných tříd a následná konfigurace, kde může dojít velmi snadno (například překlepem) k nesprávnému propojení jednotlivých komponent a jejich následné nefunkčnosti.

Při použití Fast Infosetu byl objem přenášených dat snížen na cca 15% jejich původní velikosti v XML. Měřena byla celková velikost přenášené zprávy[9] včetně SOAP obálky.

Zdrojový kód k Encoderu i testovací webové služby ve WCF je na přiloženém CD.

7.2 Efficiecnt XML Interchange

7.2.1 JAX-WS

Rozšiřitelnost JAX-WS o vlastní kódování pro přenos dat je principiálně stejná jako v případě WCF. JAX-WS definuje pipeline (na obrázku 7.2), kterou je předávána příchozí/odchozí zpráva. V rámci jednotlivých stupňů této pipeline je zpráva dekódována, je provedena autentizace a autorizace uživatele (pokud je vyžadována) a na konci pipeline dochází k zavolání příslušné webové služby.

Obrázek 7.2: Struktura JAX-WS pipeline[10]

Pro podporu vlastních formátů pro přenos dat slouží komponenta Codec, která má vlastnosti podobné Encoderu z WCF. Pro implementaci Codecu je nutné vytvořit 4 třídy, které zajišťují jeho funkcionality.

- **ExiCodec** je třída, která zajišťuje překlad mezi objektem Packet, který vystupuje z/vstupuje do pipeline a výstupním/vstupním proudem dat, která budou odeslána po přenosovém médiu. Instancím této třídy jsou také předávány jednotlivé endpointy.
- **ExiBindingId** je třídou, která uchovává unikátní identifikátor Codecu, který je následně využit v konfiguračním souboru pro specifikaci Codecu, který bude konkrétní endpoint využívat.
- **ExiBindingIDFactory** je pouze (jak název napovídá) factory, která vytváří instance ExiBindingId.

[9] Velikost přenášené zprávy byla měřena pomocí programu Wireshark, který umožňuje záznam a analýzu provozu na definovaném síťovém rozhraní.

[10] Zdroj: https://www.java.net/blog/vivekp/archive/client-pipeline.jpg

- **ExiContentType** obsahuje definici Content-Type a MIME-Type, které je daný Codec schopný zpracovat.

Poslední nutnou součástí Codecu je definice třídy, která obsahuje `BindingIDFactory`. Toho je docíleno souborem `com.sun.xml.ws.api.BindingIDFactory` v adresáři `/META-INF/services`. Obsahem tohoto souboru je definice použité BindingIDFactory - v tomto případě `com.jaxws.exi.codec.ExiBindingIDFactory`.

Nastavení nového endpointu v konfiguračním souboru `sun-jaxws.xml` vypadá následovně:

```
<endpoint
   name="CustomerSevice"
   implementation="cz.cvut.fel.dousepet.exi.CustomerService"
   url-pattern="/exi/customerService"
   binding="http://fel.cvut.cz/dousepet/codec/exi/" />
```

Po implementaci Codecu jako samostatné knihovny a vytvoření webové služby s podobnou funkcionalitou jako v případě WCF, jsem se pokoušel zprovoznit webovou službu, která by tento Codec využívala. Po odstranění očekávaných problémů s neplatnými referencemi na knihovny a špatnou konfiguraci, jsem se dostal do stavu, kdy byla při startu serveru vytvořena instance ExiCodecu, ale následně byla vyvolána blíže nespecifikovaná výjimka `"GMBAL901: JMX exception on registration of MBean MBeanImpl"` a konfigurovaný endpoint nebyl dostupný - při jeho volání byl vrácen HTTP kód 404 - Not Found.

V hledání původu vyhozené výjimky jsem nebyl úspěšný a její přesnou definici se mi také nepodařilo dohledat. Z tohoto důvodu jsem se rozhodl prozkoumat jedinou funkční uživatelskou implementaci JAX-WS Codecu, kterou se mi podařilo nalézt - jsonwebservice[11]. V tomto projektu je implementován Codec pro populární formát JSON[12]. Bohužel ani s pomocí této funkční implementace se nepodařilo zjistit, v které části jsem udělal chybu.

Přes velké množství investovaného času[13] do implementace jsem nebyl úspěšný s implementací vlastního Codecu a musel svoje snažení ukončit. Integrace Codecu pro JAX-WS tedy je možná (dokazuje projekt jsonwbservice), ale je náročnější[14] než implementace Encoderu pro WCF, protože neexistuje větší podpora pro vývojáře - největší pomocí byl blog vývojáře jsonwebservice, který popisuje postup jejího vytvoření. Tento postup se ale nepodařilo zreplikovat pro ExiCodec. Dokumentace ke API JAX-WS je dostupná ve formě JavaDocu na úrovni jednotlivých tříd, mnoho pomoci při hledání takto málo lokalizované chyby nepřináší, protože je celé API příliš rozsáhlé.

7.2.2 Windows Communication Foundation

Integrace EXI do WCF nebylo možné otestovat, protože v době vytváření této práce nebyla dostupná žádná Open Source implementace EXI pro MS .NET a zkušební verzi komerční knihovny Efficient XML se nepodařilo získat.

[11] Zdroj: <https://code.google.com/p/jsonwebservice/>
[12] JavaScript Object Notation
[13] Celkově asi 30 hodin.
[14] Hodnocení náročnosti implementace je subjektivní záležitostí, zvýšená náročnost implementace byla také ovlivněna mojí lepší znalostí a preferencí MS .NET Frameworku a souvisejících technologií.

Kapitola 8

Závěr

Cílem této práce je popsání možností na snížení komunikační a výpočetní zátěže protokolu XML, otestování vlivu vlivu jednotlivých algoritmů na výslednou velikost přenášených dat, jejich časovou a paměťovou efektivitu a rovněž možnosti jejich integrace do stávajících běžně používaných frameworků pro implementaci webových služeb.

Na splnění prvního cíle byly zaměřeny kapitoly popisující fungování algoritmů standardizovaných jako Fast Infoset a Efficient XML Interchange. Oba tyto algoritmy představují více než dobrou alternativu ke "klasickému" XML a to jak z hlediska účinnosti komprese, tak i z hlediska výkonu při kódování a dekódování obsahu. Při vzájemném porovnání mají oba algoritmy podobný základ ve formě odstraňování nadbytečných dat a stringových tabulek, ale v dalších fázích kódování se rozcházejí. Fast Infoset vychází z tohoto porovnání jako jednodušší algoritmus s větším množstvím dostupných implementací a větším rozšířením. Efficient XML Interchange naproti tomu nabízí větší možnosti konfigurace pro různé použití - například možnost přípravy dat ke kompresi, která může být provedena později (např. v aplikační vrstvě) je velice zajímavá.

Pro splnění druhého cíle byla provedena řada testů, z jejichž výsledků je zřejmé, že v mnoha případech se použití binárního kódování pro přenos dat velmi vyplatí. Velikost výsledných dat byla v případě Fast Infosetu snížena na 11% - 17%, Efficient XML Interchange dosáhlo ještě lepších výsledků se 3% - 11% z velikosti původního souboru. Také z výkonnostního hlediska vychází srovnání pro oba binární formáty lichotivě. Při porovnání platforem byl mezi Javou a MS .NETem jasným vítězem MS .NET, který byl výrazně rychlejší a paměťově úspornější. Při srovnání výkonu mezi operačními systémy nelze jednoznačně určit, která platforma byla rychlejší. Při testování interoperability jednotlivých implementací byly nalezeny menší problémy s referenční implementací Fast Infosetu v Javě a v projektu Liquid Fast Infoset pro MS .NET. S implementacemi EXI nebyly zjištěny žádné problémy, pouze je nutné mít na paměti, že EXI nezachovává prefixy původních jmenných prostorů.

Pro naplnění posledního cíle byla ověřována rozšiřitelnost populárních frameworků JAX-WS a Windows Communication Foundation. V JAX-RS byla nejdříve ověřena deklarovaná podpora Fast Infosetu, která fungovala podle očekávání a následně jsem se pokusil rozšíření služby o Codec, který zajišťuje kódování zprávy před jejím odesláním klientovi. V této fázi jsem nebyl úspěšný a vlastní implementaci se nepodařilo zprovoznit. Rozšiřitelnost webové služby ve WCF, která pro tento účel využívá Encoder, probíhala podle stejného scénáře jako v případě JAX-WS s tím rozdílem, že ve vytváření Encoderu skončilo úspěchem. Možnosti

implementace obou technologií se v tomto ohledu (rozšiřitelnost o nové možnosti kódování v transportní vrstvě) příliš neliší, ale ověření vlastní implementací se podařilo jen ve Windows Communication Foundation.

Literatura

[1] W3C Recommendation, *XML Information Set (Second Edition)*, online (2004), <http://www.w3.org/TR/xml-infoset/>

[2] W3C Recommendation, *Efficient XML Interchange (EXI)*, online (2014), <http://www.w3.org/TR/exi/>

[3] ITU-T X.891, *Fast Infoset*, online (2005), <https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=X.891>

[4] Snyder, Sheldon L., *Efficient XML Interchange (EXI) Commpression and Performance Benefits: Development, Implementation and Evaluation*, Diplomová práce (2010), Naval Postgraduate School Monterey, Kalifornie, USA

[5] Martin Gudgin (Microsoft Corporation), *Understanding Infosets*, online (2014), <http://msdn.microsoft.com/en-us/library/aa468561.aspx>

[6] Figueira, Carlos, *WCF Extensibility – Message Encoders*, online (2011), <http://blogs.msdn.com/b/carlosfigueira/archive/2011/11/09/wcf-extensibility-message-encoders.aspx>

Příloha A

Obsah CD

```
.
|-- Sources
|   |-- Java
|   |   |-- ExiCodec
|   |   |-- ExiWs
|   |   |-- ExiWsClient
|   |   |-- XmlToExi
|   |   |-- XmlToExi-Exificient
|   |   |-- XmlToFastInfoset
|   |   --- XmlToXml-DOM
|   --- NET
|       |-- FastInfoset
|       --- WcfCustomerDataProvider
--- Text
    |-- figures
    |-- cs_CZ.dic
    |-- dousepet-data.xlsx
    |-- dousepet.pdf
    |-- dousepet.tex
    |-- hyphen.tex
    --- k336_thesis_macros.sty
```