Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedry informatiky a kvantitativních metod

Porovnání formátů pro serializaci dat

Diplomová práce

Autor: Jan Vaňura
Studijní obor: Aplikovaná informatika

Vedoucí práce: Ing. Pavel Kříž, Ph.D.

Hradec Králové srpen 2017

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 8.8.2017 Jan Vaňura

Poděkování:

Děkuji vedoucímu diplomové práce Ing. Pavlu Křížovi, Ph.D. za metodické vedení práce.

Anotace

Cílem práce je porovnat formáty používané pro serializace a deserializaci dat typicky u RESTful webových služeb z hlediska jejich časové a paměťové náročnosti. Mezi testované formáty patří XML, JSON, Message Pack, Avro, Protocol Buffers a nativní serializace jednotlivých testovaných programovacích jazyků. Popsány jsou základní principy, jejich výhody a nevýhody. Serializace a deserializace jsou testovány v jazycích PHP, JAVA a JavaScript za použití oficiálních knihoven, ale i knihoven třetích stran. Výstupem práce je aplikace, která umožňuje spuštění benchmarku s proměnnými parametry. Zobrazení výsledků je jednak ve formě grafů (pro snadné porovnání) zároveň také ve formě datových CSV souborů (pro další případnou analýzu).

Annotation

Title: Comparison of data serialization formats

The aim of the thesis is to compare the formats used for serialization and deserialization of data, typically with RESTful web services, in terms of their time and memory complexity. Test formats include XML, JSON, Message Pack, Avro, Protocol Buffers, and native serialization of each of the tested programming languages. Basic principles, advantages and disadvantages of formats are described here. Serialization and deserialization is tested in PHP, JAVA and JavaScript using both official libraries as well as third-party libraries. The output of the thesis is the application that allows to run a benchmark with variable parameters. The results are graphical for easy comparison and in a form of CSV data files for further analysis.

Obsah

Seznam kódu

Seznam grafů

1 Úvod

S rozvojem informačních technologií, vzniká neustále větší důraz na výměnu informací. V dnešní době pomalu odpadá psaní tlustých desktopových aplikací. Trendem je vyvíjení takzvaných tenkých klientských aplikací, zejména webových, které vyžadují jen minimum prostředků ke spuštění na klientském počítači, často se jedná pouze o webový prohlížeč (Tynan, 2005). Dalším příklad jsou mobilní aplikace, které v posledních letech zažívají nebývalý růst. Principem těchto tenkých aplikací je v podstatě pouze zobrazování dat na straně klienta a umožnění manipulace s těmito daty skrze grafické rozhraní. Samotné operace pro změnu dat, nejrůznější výpočty nebo vyřizování žádostí o zobrazení dat se dějí na serverové straně.

Výhodou tohoto řešení je zejména snížení hardwarových nároků na klientské zařízení, aktuálnost dat pro všechny uživatele nebo například možnost naprogramování vlastní klientské aplikace (pokud je služba veřejná). Nevýhodou může být neustálá nutnost připojení k internetu, jelikož právě po této síti funguje výměna dat u naprosté většiny aplikací, pokud ovšem neumožňují práci offline.

Aby mohla taková výměna dat probíhat, je nutné, aby data byla v určitém formátu, kterému budou rozumět obě strany. Proces převedení strukturovaných dat z kódu programu do těchto formátů se nazývá serializace. Jedná se o proces transformace dat do sériové podoby. Jednoduše řečeno, do takové podoby, kdy je možné složitý objekt uložit do souboru. Opačný proces k serializaci se nazývá deserializace dat. Jedná se o proces převedení sériové struktury zpět do kódu programovacího jazyka.

Práce se zabývá porovnáním jednotlivých datových formátů z hlediska jejich paměťové náročnosti – a především jejich časové náročnosti, tedy času potřebného k serializaci a deserializaci dat. V práci jsou dále popsány principy (de)serializace do jednotlivých formátů, jejich výhody a nevýhody. Pro testování byly vybrány aktuálně nejpoužívanější datové formáty, kterými jsou XML, JSON, Avro, Message Pack a Protocol Buffers. Jedná se o otevřené standardy, čímž je umožněna implementace v takřka libovolném programovacím jazyce. Pro účely testování byly vybrány tři programovací jazyky, a sice PHP, JAVA a JavaScript. Pro úplnost byly do

testů ještě přidány nativní (de)serializace jednotlivých jazyků. Z principu tyto nativní formáty nejsou multiplatformě přenositelné. S výjimkou JavaScriptu, kde nativním formátem serializace je JSON. Z hlediska implementace konkrétního formátu většinou existuje několik způsobů, jak data serializovat či deserializovat. Proto je v testu zahrnuto několik knihoven, které danou funkčnost implementují.

Součástí práce je také aplikace, která umožňuje spuštění benchmarku. Výsledkem jsou grafy, které umožnují rychlé zhodnocení výsledků, ale i datové CSV soubory, s kterými je možné provádět další statistické výpočty.

2 Serializace

Obecně je proces serializace definován jako převedení strukturovaných dat do dat sekvenčních, která je možná uložit do souboru nebo odeslat přes síť. Strukturovanými daty jsou v terminologii objektově orientovaných jazyků myšleny především objekty tříd, které mají specifická data a jsou vzájemně provázány mezi sebou. Vztahy mezi objekty mohou být: prostá asociace nebo dědičnost, popřípadě agregace a jiné. Obecně se ale nemusí jednat pouze o objekty. Sekvenční nebo též sériová data jsou jednorozměrná data v takové formě, při které lze jednoznačně určit jejich začátek a konec; dají se uložit do souboru. Mít možnost takto data transformovat ale nestačí. Je třeba mít proces zpětné transformace, tedy převedení sériových dat zpět do strukturovaných, nejčastěji zpět do kódu programu. Tento proces se označuje jako deserializace. Snahou je oba tyto procesy co nejvíce zautomatizovat. (Surbhi, a další, 2013)

Aby v těchto procesech byl nějaký řád, vznikly nejrůznější formáty serializace dat. Některé formáty popisují pouze, jak se data budou prezentovat, jejich datové typy, syntaxi; jiné popisují i samotný proces serializace a deserializace. Více v kapitole Formáty.

Často se v souvislosti se serializací setkáváme s anglickými termíny marshalling, encode a serialize. Slova serialize a marshalling jsou někdy brána jako synonyma (Elsheimy, 2010). Serialize označuje proces serializace, který je popsaný výše. O marshallingu se nejčastěji mluví v souvislosti se vzdáleným voláním procedur (RPC[1]), posíláním vzdálených objektů (RMI[2]). Marshalling obsahuje proces serializace a navíc také tzv. codebase. Codebase obsahuje informace o tom, odkud objekt pochází, kde lze najít jeho definici nebo implementaci objektu. Slovo encode pak v překladu znamená zakódování v obecném smyslu, tedy i zakódování

[1] RPC – Remote procedure call, vzdálené volání procedur je metoda, která umožňuje programu volat kód na jiném místě, jiný PC např. v internetu.

[2] RMI – Remote Method Invocation je objektová implementace RPC v jazyku JAVA.

strukturovaných dat do sériových. Obdobně překládáme slova, která vyjadřují obrácený proces, unmarshalling, decode a deserialize. (Ryan, a další, 1999)

2.1 Druhy

Serializaci lze rozdělit podle mnoha kritérií. V dnešní době existuje mnoho serializačních formátů a samotných přístupů k serializaci. Tyto formáty se od sebe značně odlišují, přesto u nich lze pozorovat společné či rozdílné znaky, které do značné míry určují jejich použitelnost a vlastnosti. Jedním z kritérií dělení může být, jestli jsou výsledná data v binární nebo textové podobě. Dalším, zdali je vyžadováno pro serializaci schéma, které popisuje data, nebo se lze obejít bez něho.

2.1.1 Binární a textová serializace

Binární serializace je taková serializace, kdy výsledná sériová data jsou v binární podobě, tedy ve formě bitů. Taková data nejsou čitelná, ani zapisovatelná člověkem, ale obecně lze říci, že zabírají méně datového prostoru než textová (EL-BAKRY, a další, 2009).

Principem binárních formátů je většinou nejprve definice datových typů a určení jejich maximální hodnoty v bytech. Každý datový typ je určen kombinací určitých bitů, které označují začátek, popřípadě i konec. Pokud je hodnota zapsána na určitou pozici a zabírá přesně definované místo, jsme schopni zpětně tuto hodnotu zrekonstruovat (Lynx, 2014). Toto může být jeden z obecných principů, jak lze k binární serializaci přistupovat.

Jiným přístupem je textová serializace, kdy jsou sériová data zobrazena jako prosté znaky a lze si je prohlédnout v běžném textovém editoru. Výhodou těchto formátů je lidská čitelnost i zapisovatelnost. To může být výhodné například při vývoji aplikace, kdy je možné si data prohlédnout. Nevýhodou je obecně o něco větší paměťová náročnost.

2.1.2 Serializace se schématem a bez schématu

Některé formáty serializace vyžadují definici schématu předtím, než je možné přejít k samotnému procesu serializace. V tomto schématu je obvykle definována struktura dat, datové typy, názvy atributů a další dodatečné informace. Schémata

jsou obvykle psána v nějakém obecném formátu, například JSON, nebo ve vlastním formátu. Tímto způsobem je zajištěna přenositelnost mezi různými platformami. Výhodou tohoto přístupu je obvykle nižší datový objem než u formátů bez schématu. Jelikož všechna metadata nemusí být serializována, některá jsou obsažena ve schématu. Další výhodou je validace hodnot. Data jsou validní pouze, pokud vyhovují schématu. Celkově schéma popisuje hodnoty, a dává tak holým sériovým datům kontext. Nevýhodou tohoto přístupu je nutnost vytváření samotného schématu, čas potřebný k jeho napsání, popřípadě i čas potřebný k jeho nastudování. (Kreps, 2017)

Některé formáty mají definované schéma pro popis dat, přičemž schéma samotné není nutnou podmínkou pro úspěšnou serializaci.

Příkladem formátů, které vyžadují schéma, jsou Protocol Buffers nebo Avro. Formáty, které nevyžadují schéma, jsou například JSON nebo XML. Formát, který schéma vůbec nedefinuje, je MsgPack. Příkladů samozřejmě existuje více, ale výše jmenované patří mezi formáty, které budou dále testovány.

2.2 Použití

Využití serializace spočívá jednak v možnosti uložení strukturovaných dat, jednak v umožnění jejich sdílení mezi platformami, právě skrze definované formáty serializace.

2.2.1 Ukládání dat

Pokud potřebujeme ukládat malé množství dat z aplikace, například konfiguraci, můžeme k tomu využít několik metod. První nejrobustnější metodou je použití databáze. V tomto případě musíme ale zajistit mapování objektů do tabulek, využíváme-li objektově orientovaný jazyk a relační databázi. S tím nám může pomoct ORM[3] framework nebo můžeme využít objektovou databázi. Pokud se ale

[3] ORM – Object-relational mapping, do češtiny překládáno jako objektově relační mapování, je proces mapování dat z objektově orientovaných jazyků do tabulek relačních databází a zpět.

jedná skutečně jen o malá data nebo pokud nemáme k dispozici databázi, můžeme využít právě serializace a následné uložení dat do souboru. Je-li to potřeba, mohou být data ze souboru jednoduše načtena a deserializována zpět do kódu. Vybírat můžeme z několika formátů: buď textových, které umožňují jednoduchou editaci i bez aplikace, nebo binárních formátů.

Například redakční systém Wordpress, určený pro vytváření webových stránek, kombinuje obě metody, neboť do tabulky v relační databázi ukládá serializované objekty z kódu PHP. Využívá toho při ukládání uživatelského nastavení a dalších věcí. Někteří tento přístup nepovažují za nejvhodnější (Tocker , 2010).

2.2.2 Webové služby

Druhým významným použitím serializace je přenos dat u webových služeb. Dnešním trendem je vytváření tenkých klientských aplikací typu klient-server, oproti desktopovým aplikacím, a to zejména díky rozvoji chytrých mobilních zařízení či tabletů. Důvodem pro vytváření těchto aplikací je, že tato zařízení obecně nedosahují takového výkonu ani paměťového prostoru jako desktopové počítače nebo servery. Na straně klienta je vytvořena aplikace, která skrze GUI[4] umožňuje prohlížení dat, dává pokyny k manipulaci s nimi atd. Na straně serveru je aplikace, která obsahuje veškerá data a provádí s nimi většinu operací skrze předem definované aplikační rozhraní, tzv. API[5].

Komunikace mezi těmito dvě stranami často probíhá přes internet (web), pomocí tzv. webových služeb. Webové služby poskytují určité služby klientským aplikacím, které s nimi komunikují. Data takto poskytovaná klientským aplikacím jsou nejčastěji v některém z otevřených formátů, což umožňuje platformní nezávislost na obou stranách (Hostetler, a další, 2009).

---

[4] GUI – Graphical User Interface neboli grafické uživatelské rozhraní, je rozhraní, které umožňuje ovládání aplikací skrze grafické prvky (tlačítka, okna, formuláře, …).

[5] API – Application Programming Interface, aplikační rozhraní je skupina metod/chování, které program může využívat.

Mezi nejpoužívanější technologie pro webové služby patří například protokol SOAP (Simple Object Access Protocol) a popisovací jazyk WSDL (Web Services Description Language). Webová služba založená na SOAP používá ke komunikaci typicky pouze XML – pro definici operací, i pro dotazy a odpovědi. Pro komunikaci po síti SOAP využívá transportní protokoly HTTP[6], SMTP[7] nebo jiné, nejčastěji však zmíněný protokol HTTP. (Curbera, a další, 2002)

Dalším příkladem je architektura REST (Representational State Transfer). Služba založená na RESTu definuje chování pomocí HTTP protokolu. Konkrétně využívá čtyři HTTP metody: POST pro vložení záznamu, GET pro požadavek na data, PUT pro editaci dat a DELETE pro smazání. Formát, ve kterém probíhá komunikace, je libovolný. Nejčastěji se používá JSON nebo XML. (Richardson, a další, 2007)

Dalším příkladem jsou RPC (Remote procedure call) služby či vzdálené volání procedur. Obdobně jako webové služby RPC umožňuje komunikaci mezi dvěma oddělenými aplikacemi. V jistém smyslu jsou webové služby implementací RPC (Trautmann, 2012). Příkladem je například gRPC, framework určený k RPC komunikaci pro formát Protocol Buffers (Protobuf); vlastní RPC také definuje Avro formát a další, viz kapitoly dále.

[6] HTTP – Hypertext Transfer Protocol je bezstavový protokol využívající k distribuci hypertextového obsahu, nejčastěji webových stránek.

[7] SMTP – Simple Mail Transfer Protocol je protokol pro výměnu elektronické pošty.

3 Formáty

Jednou z nejdůležitějších otázek při serializaci je výběr formátu, do kterého budeme serializovat. Tedy jakou podobu budou mít naše výsledná sériová data. Záleží nám pouze na rychlosti zpracování, velikosti výsledných dat, nebo je důležitá přenositelnost formátu? Popřípadě: je nutné, aby byla data lidsky čitelná (zapisovatelná)?

Na otázku rychlosti a velikosti dat odpoví tato práce v pozdějších kapitolách. Co se týká přenositelnosti, většina dnes používaných formátů má otevřenou specifikaci a existuje hned několik implementací do nejrůznějších programovacích jazyků. Přesto existuje několik formátů, které toto pravidlo nesplňují. Jedná se především o nativní serializace samotných programovacích jazyků. Některé z nich jsou popsány v pozdějších kapitolách. V poslední řadě se jedná o lidskou čitelnost (zapisovatelnost) dat. Jak již bylo řečeno v kapitole o serializaci, máme zde dvě hlavní kategorie. Textovou a binární serializaci, tedy textový nebo binární formát, přičemž lidsky čitelný je ten textový.

V následujících kapitolách je představen přehled dnes nejpoužívanějších formátů. Popsány jsou obecné principy, syntaxe, možné výhody a nevýhody daného formátu.

3.1 XML

Jedním z nejdéle používaných formátů pro serializaci dat je XML (eXtensible Markup Language). Do češtiny se překládá jako rozšiřitelný značkovací jazyk. Jedná se o jeden z nejrozšířenějších formátů, který můžeme najít v mnoha odvětvích IT, například u zdravotnických systémů, GPS systémů a dalších (Fawcett, a další, 2012). Za vznikem tohoto jazyka stojí konsorcium W3C[8]. XML bylo obdobně jako HTML vyvinutu ze staršího jazyka SGML (Standard Generalized Markup Language). Na rozdíl od HTML, jež obsahuje hotovou definici elementů, které je možné použít, XML pouze definuje sadu pravidel pro tvorbu vlastních značek (elementů). Proto bylo

[8] W3C – World Wide Web Consortium je organizace, které vyvíjí webové standardy, jako HTML, XML a další.

použito písmeno X jako eXtensible (rozšířitelný) v názvu. Jedná se o textový formát, který je snadno čitelný i zapisovatelný člověkem. Ukázka XML dokumentu (viz Kód 1):

Kód 1 - XML dokument

```
<?xml version="1.0" encoding="UTF-8"?>
<persons>
  <!-- Toto je komentář XML -->
  <person>
    <name>Gates Ochoa</name>
    <born format="yyyy-mm-dd">1990-01-05</born>
    <email>gates@ochoa.cz</email>
    <about>Lorem ipsum dolor sit amet, consectetur adipiscing elit.</about>
  </person>
  <person>
    <name>Alison Rosales</name>
    <born format="yyyy-mm-dd">1990-05-01</born>
    <email>alison@rosales.cz</email>
    <about>Aenean vel orci dignissim, consequat diam vitae, rhoncus
quam.</about>
  </person>
</persons>
```

Každý XML dokument by měl začínat deklarací verze a použité znakové sady, jak je vidět na ukázce (viz Kód 1) na prvním řádku. Co se týká verzí XML, v současnosti existují dvě verze, verze 1.0 a 1.1. Doporučeno je používat verzi 1.0, od verze 1.1 se liší v některých aspektech jako rozšíření podporované znakové sady a whitespace znaků. Verze 1.0 se nadále vyvíjí, aktuálně je k dispozici pátá edice (rok 2008), ovšem bez změny označení. (Harold, 2003)

Dalším pravidlem je, že každý XML dokument by měl obsahovat právě jeden kořenový element, zbylé elementy musí být zanořené uvnitř tohoto elementu. Elementy mohou být párové nebo nepárové a mohou obsahovat text, atribut nebo jiný element. Každý z elementů, ale musí být uzavřen a nesmí se křížit s jiným. Na samotné názvy elementů jsou také kladena pravidla (Bray, a další, 2008):

- Názvy elementů jsou citlivé na velikost písmen.
- Název elementu musí začínat písmenem nebo podtržítkem.

- Název elementu nesmí začínat písmeny xml nebo XML a jejich dalšími variantami.
- Název elementu může obsahovat písmena, čísla, pomlčky, podtržítka a tečky.
- Název elementu nesmí obsahovat mezeru.

Následuje ukázka správně a špatně použité syntaxe XML, viz Kód 2.

Kód 2 - XML nevalidní elementy

```
<0person></0person>
<xmlPerson></xmlPerson>
<person 0></person 0>
<person+0></person+0>
<person0><person1></person0></person1>
```

Dalším omezením, na které je potřeba si dát pozor, je použití speciálních znaků uvnitř elementů. Tyto znaky je potřeba nahradit příslušnými entitami. XML definuje pět těchto entit, a sice *< (<), > (>), & (&), ' ($apos;), " (")*. Pouze znaky < a & generují chybu při zpracování XML, i přesto je vhodné nahrazovat zbylé znaky příslušnými entitami. Existuje ještě jeden způsob, díky kterému nemusíme nahrazovat tyto speciální znaky, a sice použití tzv. CDATA. Pokud je obsah elementu obalen pomocí takového zápisu, pak se tato data nebudou interpretovat jako XML syntaxe. (Kosek, 2000) Viz ukázku Kód 1.

Kód 3 - XML CDATA

```
<about><![CDATA[Aenean <b>vel orci</b> dignissim, consequat diam.]]></about>
```

Stejná pravidla ohledně tvorby názvů elementů se vztahují i na názvy atributů. Zároveň hodnota atributu musí být uvozena v uvozovkách, čímž se liší zápis XML atributů od HTML, kde lze použít jednoduché uvozovky nebo je v některých případech zcela vynechat. Atributy, nám dávají druhou možnost, jak zapsat data do XML. Kdy tedy použít atribut a kdy zapsat hodnotu do elementu? Odpověď na tuto otázku není jednoznačná a víceméně záleží na programátorovi, jak se tohoto problému chopí. Obecně se dá postupovat podle následujícího pravidla: pokud jsou data relevantní pro program, který XML zpracovává, pak se doporučuje použít atribut, v opačném případě element.

Obecné XML neobsahuje žádné datové typy, jedná se o holý text. Zejména u použití ve webových službách by však bylo vhodné, aby program měl možnost validovat XML, a to z hlediska datových typů, ale i samotné struktury. I na to formát XML myslí a dává nám možnost, jak tohoto chování dosáhnout. V zásadě existují dvě řešení:

- DTD (Document Type Definition).
- XSD (XML Schema Definition).

První z uvedených způsobů umožňuje deklarovat výčet názvů elementů, přípustný počet opakování, definici atributů a další. DTD ale neumožňuje použít definici datových typů jako number, boolean aj, právě z toho důvodu není často využíván a je nahrazován novějším XSD. Samotná definice může být součástí XML nebo obsažena v externím souboru s příponou dtd. V obou případech se pro připojení k XML využívá tag *!DOCTYPE*. (Kosek, 2000) Ukázka (viz Kód 4):

Kód 4 - XML DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE persons [
  <!ELEMENT persons (person*)>
  <!ELEMENT person (name,born,email,about)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT born (#PCDATA)>
  <!ATTLIST born format CDATA "yyyy-mm-dd">
  <!ELEMENT email (#PCDATA)>
  <!ELEMENT about (#PCDATA)>
]>
<persons>
  <person>
    <name>Pepa Zdepa</name>
    <born format="yyyy-mm-dd">1990-01-05</born>
    <email>pepa@zdepa.cz</email>
    <about>Lorem ipsum dolor sit amet, consectetur adipiscing elit.</about>
  </person>
</persons>
```

Jak již bylo řečeno, DTD je nahrazováno novějším standardem pro definici schémat, tzv. XSD. V podstatě se jedná o XML, které popisuje XML. Na rozdíl od běžného XML toto schéma již obsahuje základní sadu elementů a atributů, pomocí kterých lze XML

dokumenty popisovat. Každý z těchto elementů nebo atributů má přesně daný význam. Tato schémata se dají také rozšiřovat, lze tvořit vlastní datové typy, výčtové typy (enumerace), validace oproti regulárním výrazům a další. Jedno schéma je možné použít pro validaci vícero XML dokumentů. V jednom XML lze využít více schémat najednou. Všemi těmito vlastnostmi se XSD liší od DTD, které toto chování nepodporuje. Největší výhodou ale zůstává možnost definování primitivních datových typů (*string, integer, date, …*), které u DTD zcela chybí. Pro odkazování na konkrétní schéma se používají tzv. jmenné prostory. Ukázka (viz Kód 5):

Kód 5 - XML XSD

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="https://www.janvanura.cz"
xmlns="https://www.janvanura.cz"
elementFormDefault="qualified">

<xs:element name="persons">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="person"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="born" type="xs:string">
        <xs:complexType>
          <xs:attribute name="format" type="xs:string" use="required"
default="yyyy-mm-dd"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="email" type="emailAddress"/>
      <xs:element name="about" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:simpleType name="emailAddress">
  <xs:restriction base="xsd:string">
      <xs:pattern value="([0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*@([0-9a-zA-Z][-\w]*[0-
9a-zA-Z]\.)+[a-zA-Z]{2,9})"/>
  </xs:restriction>
</xs:simpleType>
</xs:schema>
```

V ukázce označené Kód 5 je vidět použití jmenného prostoru http://www.w3.org/2001/XMLSchema, který je prefixovaný řetězcem *xs*. Každý element v tomto schématu začíná názvem *xs:*. Pokud bychom používali více schémat,

pak právě tímto prefixováním můžeme rozlišovat elementy. V ukázce je dále vidět odkazování na element *person* nebo například definice vlastního typu *emailAddress* a další. (Kosek, 2000)

K XML vzniklo několik podpůrných technologií, které usnadňují práci s tímto formátem. Jedním, který stojí za zmínku, je jazyk XPATH (XML Path Language). Tento jazyk umožňuje vybírat části XML dokumentu pomocí zadaných kritérií (například názvu elementu/atributů, jejich pořadí, obsahu aj.) obdobně jako CSS selektory v HTML. Další zajímavou technologií je XSLT (eXtensible Stylesheet Language Transformations). Tento proces umožňuje transformaci XML dokumentu do jiného formátu (nejčastěji HTML) pomocí pravidel definovaných v XSL souboru (opět se jedná o XML). (Bray, a další, 2008)

K zpracování XML dokumentů v programu existují dva hlavní přístupy, a sice DOM (Document Object Model) a SAX (Simple API for XML Parsing) parser. Principem DOM parseru je načtení celého dokumentu do paměti a převedení na stromovou strukturu, ve které se lze pohybovat například pomocí XPATH jazyka. Naproti tomu SAX parser nenačítá celý dokument do paměti, ale pouze vyvolává události při jeho postupném čtení, například začátek elementu, konec elementu atd. SAX přístup je zejména výhodný při zpracování velkých XML souborů, kde nám nemusí stačit operační paměť. (Kosek, 2000)

Obecně tedy výhodou XML je jeho velká rozšiřitelnost napříč nejrůznějšími platformami, jednoduchost (lidsky čitelný i zapisovatelný), možnost validace pomocí schémat, velké množství podpůrných technologií. K nevýhodám formátu ale patří jeho velká „upovídanost“ (Maeda, 2011). To znamená, že velkou část datového objemu XML zabírají názvy elementů, které se navíc častokrát opakují (začáteční a koncová značka). Struktura, která z hlediska relevance dat není důležitá, může zabírat nejvíce velikosti z celkového objemu dat. Zejména pokud XML obsahuje malá (krátká) data v obsahu elementů a naopak dlouhé názvy těchto elementů.

3.2 JSON

Dalším velice rozšířeným a hojně využívaným formátem je JSON (JavaScript Object Notation). Jak název napovídá, tento formát vychází z JavaScriptové notace zápisu

objektů. Jedná se o lehký formát, který častokrát nahrazuje XML (kvůli jeho upovídanosti), zejména u webových služeb typu REST (Richardson, a další, 2007). Obdobně jako u XML se jedná o textový formát, který je lehce čitelný člověkem. Zapisovatelnost člověkem již není tak intuitivní jako XML, jelikož syntaxe JSON používá sadu závorek, čárek a uvozovek, ve kterých (bez použití IDE[9]) lze jednoduše udělat chybu.

Na rozdíl od XML, kde vše je pouhým textem, JSON definuje základní datové struktury jako *object* (objekt), *array* (pole) a základní datové typy jako *string* (řetězec), *number* (číslo), *boolean* (logická hodnota) a *null* (nedefinovaná hodnota). (Bray, 2014)

Každý JSON dokument musí začínat buď definicí objektu, nebo pole. Zbylá data se musí nacházet uvnitř této struktury obdobně jako v kořenovém elementu u XML. Objekty i pole se mohou libovolně zanořovat do sebe, čímž tvoří danou strukturu. Hodnoty typu *string* musí být uvozeny ve dvojitých uvozovkách, přičemž typ *number* nikoliv. Hodnoty *true, false, null* musí být psány malými písmeny. Pole je definováno pomocí hranatých závorek a prvky v poli se oddělují pomocí čárky. Za posledním prvkem čárka být nesmí. Naproti tomu objekty se uvozují do složených závorek. Prvky objektů musí být zapsány stylem klíč, dvojtečka, hodnota (jedná se v podstatě o hash-mapu) a oddělují se čárkou stejně jako prvky pole. Klíč prvku objektu musí být uvozen ve dvojitých uvozovkách. Pokud se v klíči nebo jiné *string* hodnotě vyskytuje znak dvojité uvozovky, musí být tato hodnota escapována zpětným lomítkem. (Bray, 2014) Následuje ukázka viz Kód 6.

[9] IDE – Integrated Development Environment, vývojové prostředí je program, který pomáhá při vyvíjení aplikací.

Kód 6 - JSON dokument

```
{
"persons": [
    {
      "name": "Gates Ochoa",
      "born": {
        "format": "yyyy-mm-dd",
        "value": "1990-01-05"
       },
       "email": "gates@ochoa.cz",
       "about": "Lorem ipsum dolor sit amet, consectetur adipiscing elit."
    },
    {
      "name": "Alison Rosales",
      "born": {
        "format": "yyyy-mm-dd",
        "value": "1990-05-01"
       },
       "email": "alison@rosales.cz",
       "about": "Aenean vel orci dignissim, consequat \"diam\" vitae, rhoncus
quam."
    }
  ]
}
```

Použité formátování viz Kód 6 v podobě odsazení řádků nemá žádný funkční význam a slouží pouze k lepšímu čtení formátu.

Pro JSON stejně jako pro XML existuje nástroj pro validaci dokumentů, tzv. JSON Schema. Jedná se o JSON dokument, který popisuje, jak má vypadat JSON dokument. Toto schéma umožňuje validovat JSON z hlediska datových typů proměnných, jejich povinnost výskytu a další. Od JSON přebírá základní datové typy, navíc je rozšiřuje o typ *integer* (celé číslo). Dále definuje již několik předpřipravených validátorů jako email, URL, datum a jiné. Schémata lze různě kombinovat a vkládat do sebe a poté pomocí nich validovat více JSON dokumentů. Základní JSON metaschema obsahuje právě předešlé zmíněné struktury. Schéma pro systém odkazování na jiná schémata

používá URI[10]. Zaměřit lze i menší sekci než jen celá schémata, identifikované pomocí URI, ale i jednotlivá podschémata. K tomuto účelu se používá URI, za níž následuje znak # a název podschématu, obdobně jako u odkazování na kotvy v HTML dokumentu (Droettboom, 2016). Ukázka JSON schématu, které validuje Kód 6, lze najít viz Kód 7.

[10] URI – Uniform Resource Identifier, jednoznačný identifikátor zdroje je obecně řetězec využívající v informatice pro jednoznačnou identifikaci

Kód 7 - JSON Schema

```
{
    "id": "https://www.janvanura.cz",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "persons": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "name": {"type": "string"},
                    "born": {
                        "type": "object",
                        "properties": {
                            "format": {"type": "string"},
                            "value": {"type": "string"}
                        },
                        "required": ["format", "value"]
                    },
                    "email": {
                        "type": "string",
                        "format": "email"
                    },
                    "about": {"type": "string"}
                },
                "required": ["name", "born", "email", "about"]
            }
        }
    },
    "required": ["persons"]
}
```

Toto schéma je vhodné například na validaci dotazů a odpovědí k webové službě. Existují i nástroje, které z těchto schémat dokáží generovat dokumentaci k API. Nevýhodou tohoto schématu je samotný fakt, že je psáno v JSON formátu. Jak již bylo řečeno, JSON nebyl navržen proto, aby byl lidsky zapisovatelný. Zejména u komplikovanějších struktur, kde je potřeba dávat si pozor na velké množství závorek, čárek, dvojteček, uvozovek atd., je velice jednoduché udělat chybu.

JSON formát se někdy považuje za odlehčenou variantu XML. V některých případech jej také začal nahrazovat (Pragmateek, 2013), například při přenosu dat v asynchronních požadavcích na webu, tzv. AJAX (Asynchronous JavaScript and XML). Tyto požadavky jsou nejčastěji obsluhovány pomocí JavaScriptu, kde je JSON pohodlnější zpracovávat v porovnání s XML. Z toho důvodu se někdy tato technologie označuje zkratkou AJAJ (Asynchronous JavaScript and JSON).

K celkovým výhodám JSON patří jeho jednoduchost a textový, lidsky čitelný formát. Samotné struktury používané v JSON (objekt, pole) napovídají do jakých struktur má být dokument deserializován v použitém jazyce (Strassner, 2015). Jelikož většina dnes používaných jazyků obsahuje struktury typu objekt nebo pole, popřípadě jiné varianty jako *records, structs, hash map, list* atd. Jistou nevýhodou může být, že JSON neumožňuje nativní zápis atributů, které mohou být využity pro metadata, obdobně jako v XML. K tomuto účelu se někdy používá notace pomocí zavináče (@), přičemž název klíče prefixovaný zavináčem značí atribut (Newton-King, 2015).

3.3 Avro

Dalším z testovaných formátů je Avro. Tento formát pro serializaci dat je vyvíjen pod záštitou organizace svobodného softwaru Apache. Celým názvem tedy Apache Avro. Oficiálně je podporováno okolo deseti programovacích jazyků, ve kterých je Avro implementováno (C, C++, C#, JAVA, JS, PERL, PHP, PYTHON, RUBY). Dále existují neoficiální implementace do dalších jazyků jako například HASKELL (DuBuisson, 2017).

Na rozdíl od dvou předchozích formátů JSON a XML je Avro binární formát. Po procesu serializace jsou data zakódována do binární podoby a nelze je „lidsky“ číst ani zapisovat. Pro účely vývoje a testování umožňuje Avro serializaci do JSON formátu. Toto chování ale není implementováno ve všech jazycích. Dalším rozdílem je, že Avro vyžaduje definici schématu. Toto schéma je třeba vždy definovat předtím, než je možné přistoupit k serializaci dat. Samotné schéma je definované pomocí JSON formátu. Avro ve své specifikaci podporuje nejen (de)serializaci dat, ale také

umožňuje generovat kód z vytvořeného schématu. Dále také popisuje vzdálenou komunikaci, tzv. vzdálené volání procedur RPC. (Blue, 2017)

Tato dvě nadstandardní chování nejsou implementována ve všech jazycích. Například pro jazyk PHP existuje pouze implementace serializace a deserializace dat. Navíc data musí být zapsána v polích (array). U PHP dále chybí implementace serializace tříd a tudíž i generování ze schémat i vzdálené RPC volání.

Avro schéma je v principu obdobné JSON Schema. Stejně jako JSON Schema definuje: jak budou data vypadat, názvy proměnných, datové typy, přípustné hodnoty atd. Avro schéma definuje osm základních primitivních datových typů (Blue, 2017):

- Null – nedefinovaná hodnota.
- Boolean – logická hodnota.
- Int – celé číslo, přesnost 32 bitů.
- Long – celé číslo, přesnost 64 bitů.
- Float – desetinné číslo, přesnost 32 bitů, standard IEEE 754.
- Double – desetinné číslo, přesnost 64 bitů, standard IEEE 754.
- Bytes – sekvence 8 bitů.
- String – sekvence znaků v unicode.

Dále šest komplexních datových typů:

- Record – analogie s objektem.
- Enums – výčtový datový typ.
- Arrays – pole prvků.
- Map – pole prvků typu klíč hodnota.
- Unions – použit, pokud má prvek více datových typů, například řetězec a číslo.
- Fixed – pro prvky stejné velikosti v bytech.

Ukázka Avro schéma, které definuje data viz Kód 6 lze najít viz Kód 8.

Kód 8 - Avro schéma

```
{
    "name": "data",
    "type": "record",
    "fields": [{
        "name": "persons",
        "type": {
            "type": "array",
            "items": {
                "name": "persons_rec",
                "type": "record",
                "fields": [
                    {"name": "name", "type": "string"},
                    {"name": "born", "type": "int", "logicalType": "date"},
                    {"name": "email", "type": "string"},
                    {"name": "about", "type": "string"}
                ]
            }
        }
    }]
}
```

Jak již bylo řečeno, z těchto schémat lze generovat kód, například třídy objektů, ze kterých se provádí následná (de)serializace dat. Není to však nezbytné, neboť Avro dokáže pracovat i bez tohoto generovaného kódu (v některých implementacích není součástí). Pokud jsou data serializována do souboru, je k nim přiloženo i samotné schéma. (Blue, 2017)

Vzdálené volání procedur (RPC) je definováno opět pomocí JSON formátu. V tomto dokumentu jsou definovány základní popisné charakteristiky komunikace jako název protokolu, typy zpráv, možné odpovědi a požadavky a další. Samotná komunikace pak může probíhat například pomocí HTTP (Hypertext Transfer Protocol) protokolu nebo jiných. Před samotným požadavkem a odpovědí ještě probíhá proces zvaný Handshake (podání rukou). Během tohoto procesu se zajišťuje, aby obě strany (server i klient) měly definici protokolu, a mohla tak správně probíhat serializace a deserializace zpráv (Blue, 2017). Ukázka Avro RPC protokolu viz Kód 9.

Kód 9 - Avro RPC protokol

(zdroj: https://avro.apache.org/docs/1.8.2/spec.html)

```
{
  "namespace": "com.acme",
  "protocol": "HelloWorld",
  "doc": "Protocol Greetings",
  "types": [
    {"name": "Greeting", "type": "record", "fields": [
      {"name": "message", "type": "string"}]},
    {"name": "Curse", "type": "error", "fields": [
      {"name": "message", "type": "string"}]}
  ],
  "messages": {
    "hello": {
      "doc": "Say hello.",
      "request": [{"name": "greeting", "type": "Greeting" }],
      "response": "Greeting",
      "errors": ["Curse"]
    }
  }
}
```

Výhodou formátu Avro je jeho schéma, které lze do jisté míry vnímat jako nevýhodu. Výhodami jsou jednoznačně daný popis, názvy proměnných, přípustné hodnoty atd. Díky schématu může Avro jednoduše validovat příchozí a odchozí zprávy nezávisle na zvoleném jazyku, popřípadě generovat kód. Nevýhodou Avro schématu je zvolený formát, a to JSON. Zejména u složitějších struktur je JSON formát obtížně zapisovatelný člověkem (nutnost dávat si pozor na uvozovky, závorky atd.). Další výhodou Avro je, že obsahuje specifikaci pro PRC. Nevýhodou této specifikace je opět JSON formát a relativně malá podpora implementace napříč technologiemi.

3.4 Protocol Buffers (Protobuf)

Protocol Buffers, zkráceně Protobuf, je formát serializace vyvíjený firmou Google. Obdobně jako Avro formát se jedná o binární formát, který ke své práci vyžaduje definované schéma. Z definici schématu jsou následně vygenerovány datové struktury pro konkrétní programovací jazyk. Stejně jako u ostatních testovaných formátů, Protobuf podporuje mnoho programovacích jazyků.

Pro definici schématu protobuf používá vlastní formát, a sice tzv. proto zprávy. Zprávy jsou koncipovány obdobně jako třídy v objektovém programování. Zpráva obsahuje název a výčet atributů včetně datových typů. Následně ještě může obsahovat některé nepovinné údaje jako název balíčku, název vygenerované struktury a další. Atributy mohou být povinné, volitelné, opakující nebo typu enumerace (výčtový typ). Dále je třeba definovat typ atributu. Protobuf podporuje několik základních datových typů jako: *bool, string, bytes, double, float, int32, int64* aj. (Skeet, 2017)

Jako datový typ lze použít i jinou zprávu, která může být definována ve stejném nebo jiném proto souboru. V neposlední řadě je třeba definovat pořadí atributu. Jedná se o číslo, které určuje pořadí dat v serializované podobě. Ukázku proto zprávy je možné vidět viz Kód 10.

Kód 10 - Protobuf definice proto zprávy

```
syntax = "proto3";

import "google/protobuf/timestamp.proto";

message PersonCollection {
  // some comment...
  repeated Person persons = 1;
}

message Person {
  required string name = 1;
  required google.protobuf.Timestamp born = 2;
  required string email = 3;
  optional string about = 4;
}
```

Z proto zpráv se následně vygeneruje datová struktura pro konkrétní jazyk, například třídy pro jazyk JAVA. Generování zajišťuje tzv. Protocol Buffers Compiler, zkráceně protoc. Tento nástroj je součástí oficiální distribuce. Vygenerovaný kód obsahuje základní metody pro práci jako settery, gettery, metody pro serializaci a deserializaci a další. Po vygenerování kódu se již s proto soubory nepracuje a nejsou zapotřebí ani pro samotnou serializaci a deserializaci. Touto vlastností se liší od formátu Avro, kde je nutné schéma vždy načítat.

Pro Protobuf také existuje definice vzdáleného volání procedur RPC. Tato podpora ale není součástí samotné definice Protobuf, ale vznikla jako vedlejší projekt. Jedná se o tzv. gRPC (Tiller, 2007).

Mezi výhody Protobuf formátu patří jeho schéma, které zajišťuje jednoznačnou definici dat. Podporována je většina hlavních programovacích jazyků včetně generování kódu, které je nezbytnou součástí Protobuf. Vlastní formát pro definici schématu je jednoduše zapisovatelný i u složitějších struktur, staví na principech objektově orientovaného programování. Jistou nevýhodou tohoto formátu je rychlost nasazení, jelikož je třeba nejprve vytvořit schéma, poté vygenerovat kód a až následně je možné serializaci využívat.

3.5 Message Pack (MsgPack)

Dalším z formátů serializace je Message Pack, zkráceně MsgPack. Jedná se o binární formát obdobně jako Protobuf nebo Avro, ale na rozdíl od těchto formátů nevyžaduje žádné schéma. Jelikož je binární, odpadá lidská čitelnost či zapisovatelnost. MsgPack je velice rozšířený z hlediska podpory nejrůznějších jazyků. Můžeme najít implementace od Javy přes Haskell až po Arduino C (Furuhashi, 2015).

MsgPack obsahuje podporu několika datových typů. Jedná se o *Integer, Nil, Boolean, Float, String, Binary, Array, Map* a *Extension.* Nicméně i datové typy mají svá omezení. Hodnoty pro celá čísla (*Integer*) jsou limitovány od -2^{63} do $2^{64}-1$. Desetinná čísla (*Float*) jsou ukládána podle standardu IEEE 754, který umožňuje uložit $(2-2^{-23}) \times 2127 \approx 3.402823 \times 1038$ čísel. Maximální délka binárních dat (*Binary*) je $2^{32}-1$ bytů. Stejně tak maximální délka řetězce (*String*) je omezena na $2^{32}-1$ bytů. Omezení se vztahují i na pole (*Array*) a asociativní pole (*Map*), v obou případech je počet prvků maximálně $2^{32}-1$. MsgPack umožňuje definovat vlastní datový typ skrze typ *Extension*. Tento datový typ se skládá ze dvou částí. První částí je celé číslo (*Integer*), které slouží pro identifikaci typu, a pole bytů, které slouží pro samotnou hodnotu. Aplikace umožňuje definovat až 128 vlastním datových typů, od 0 do 127. (Furuhashi, 2015)

Jako hlavní rival tohoto formátu se často označuje JSON. Obdobně jako JSON míří na jednoduché použití a malou velikost dat. V kódu obvykle stačí zavolat pouze jedinou funkci, která provede serializaci bez nutnosti zásahu do okolního kódu (samozřejmě záleží na zvoleném jazyce a implementaci). Obdobně to platí pro deserializaci. Co se týká velikosti, data jsou skutečně menší než v případě JSON, jak dokládá i provedený benchmark. Ukázka viz Kód 11.

Kód 11 - MsgPack vs JSON

```
{
"persons": [
    {
      "name": "Gates Ochoa",
      "born": {
        "format": "yyyy-mm-dd",
        "value": "1990-01-05"
       },
       "email": "gates@ochoa.cz",
       "about": "Lorem ipsum dolor sit amet, consectetur adipiscing elit."
    }
  ]
}
```

```
81 a7 70 65 72 73 6f 6e 73 91 84 a4 6e 61 6d 65 ac 47 61 74 65 73 20 4f 63 68
6f 61 20 a4 62 6f 72 6e 82 a6 66 6f 72 6d 61 74 aa 79 79 79 79 2d 6d 6d 2d 64
64 a5 76 61 6c 75 65 aa 31 39 39 30 2d 30 31 2d 30 35 a5 65 6d 61 69 6c ae 67
61 74 65 73 40 6f 63 68 6f 61 2e 63 7a a5 61 62 6f 75 74 d9 38 4c 6f 72 65 6d 20
69 70 73 75 6d 20 64 6f 6c 6f 72 20 73 69 74 20 61 6d 65 74 2c 20 63 6f 6e 73
65 63 74 65 74 75 72 20 61 64 69 70 69 73 63 69 6e 67 20 65 6c 69 74 2e
```

Velikost serializovaných dat viz Kód 11 v případě JSON činí 258 bytů, pokud odebereme veškeré bílé znaky, které slouží pouze pro lepší čitelnost, dostaneme se na 184 bytů. V případě MsgPack ta samá data mají pouze 154 bytů, to znamená pouze 59 % původních dat, popřípadě 84 % při odstranění bílých znaků. V případě použití Avra nebo Protobuf bude výsledek s největší pravděpodobností ještě menší, jelikož tyto formáty vyžadují schéma, a tudíž část informací nemusí být přítomna v serializovaných datech.

Výhodou formátu MsgPack je stejně jako u předchozích formátů jeho rozšiřitelnost napříč platformami. Dále také jednoduchost použití, neboť není potřeba definovat schéma. Tato výhoda může být ale i jistou nevýhodou, jelikož MsgPack nepodporuje žádnou validaci dat pomocí externího schématu obdobně jako JSON nebo XML. Není tedy možné klást omezující podmínky na serializovaná data, definovat datové typy atd. Například v jazyce Perl může dojít ke špatné serializaci čísla jako řetězce, jelikož Perl mezi těmito typy přímo nerozlišuje. Bez schématu také nemůžeme platformě nezávisle validovat požadavky a odpovědi webové služby.

3.6 Nativní formáty

Velká spousta programovacích jazyků obsahuje vlastní formáty serializace. Výhodou tohoto přístupu je jednoduché použití, není totiž zapotřebí připojovat knihovny třetích stran. Dále bývá také zajištěna zpětná kompatibilita napříč verzemi konkrétního jazyka, jelikož formát serializace se jen zřídkakdy mění tak, aby způsobil zpětnou nekompatibilitu. Zjevnou nevýhodou ale je nepřenositelnost mezi jinými jazyky, výjimkou je případ, kdy existuje implementace nativní serializace jednoho jazyka do druhého. Právě z hlediska obtížné přenositelnosti nejsou tyto serializace využívány u webových služeb, kde klientské aplikace mohou být napsány v jiném jazyce než serverová část. Ovšem pro použití jednoduchého ukládání dat, pokud například nechceme použít databázi nebo není k dispozici, se jedná o velice šikovné a bezstarostné řešení.

3.6.1 PHP

PHP od verze 4 obsahuje podporu pro nativní serializaci datových struktur. Serializovat lze primitivní datové typy (*int, float, string, boolean*), pole i objekty tříd. Jedinou výjimkou je typ *resource*, který nelze serializovat. Jedná se o speciální proměnnou, která drží referenci na externí zdroj, jakým je například otevřený soubor, databázové spojení atd. (Achour, 2017)

Serializace je provedena zavoláním funkce *serialize()*. Tato funkce jako parametr přijímá proměnnou obsahující jakýkoliv z výše uvedených typů kromě *resource*. Návratová hodnota funkce je typu řetězec (*string*). Jedná se o sekvenci bytů, kterou je možné uložit kamkoliv, například soubor nebo databáze. V případě ukládání do

relační databáze by měl být použit typ *BLOB* oproti běžným typům jako *char* nebo *text*. To z toho důvodu, že řetězec může obsahovat *nul* byty, které by se mohly při ukládání/načítání špatně interpretovat. (Achour, 2017)

Následuje praktická ukázka použití funkce *serialize()* na různých vstupních hodnotách viz Kód 12.

Kód 12 - PHP funkce serialize()

(zdroj: http://www.phpinternalsbook.com/classes_objects/serialization.html)

```php
serialize(1);
// výstup -> i:1;

serialize(2.3547);
// výstup -> d:2.3546999999999998;
//             ^-- přesnost lze ovlivnit serialize_precision direktivou v
php.ini

serialize("string");
// výstup -> s:6:"string";
//             ^-- délka řetězce strlen()

serialize(true);
// výstup -> b:1;

serialize(false);
// výstup -> b:0;

serialize(null);
// výstup -> N;

serialize([1 => "value", "key" => 0]);
// výstup -> a:2:{i:1;s:5:"value";s:3:"key";i:0;}
//             ^-- počet prvků pole count(), poté následuje {klíč;hodnota;...}

$reference = ["foo"];
$reference[1] =& $reference[0];
serialize($reference);
```

Důležitou částí je zde R:2;. V překladu to v podstatě znamená reference na druhou hodnotu. Co je druhá hodnota? První hodnotou je celé pole, prvním indexem pole je

(s:3:“foo“;), které je druhou hodnotou, tam tedy reference odkazuje (Pauli, a další, 2013).

Opačnou funkcí k funkci *serialize()* je funkce *unserialize()*. Tedy proces obnovení serializovaných dat zpět do kódu programu. Funkce přijímá dva parametry. Prvním je serializovaný řetězec, tento parametr je povinný. Druhým nepovinným parametrem lze určit, jaké třídy se budou serializovat, tzv. whitelist. Vstupem je pole obsahující jména povolených tříd. Jako vstup lze také uvést logické hodnoty *true* nebo *false*. Při hodnotě *false* nebude docházet k žádné serializaci tříd, ale pouze primitivních datových typů a polí. Uvedením hodnoty *true* říkáme serializuj všechny třídy bez rozdílu, stejné jako bez uvedení parametru. Funkce vrací PHP hodnotu odpovídající serializovanému řetězci. (Achour, 2017)

Při serializaci objektů se navíc volají dvě magické metody *__sleep()* a *__wakeup()*. Metoda *__sleep()* se volá jednou při serializaci objektu a měla by vracet pole obsahující názvy proměnných, které chceme serializovat. Dále je možné v této metodě provádět úklid před serializací. Nevýhodou této funkce je, že není možné deklarovat proměnnou, která je definována v předkovi. Obdobně při zavolání funkce unserialize() nad objektem je zavolána metoda *__wakeup()*. Tato metoda nemusí nic vracet a slouží například k obnovení reference na nějaký zdroj (*resource*) (Achour, 2017). Následuje praktické ukázka použití funkce *serialize()* nad objektem viz Kód 13.

Kód 13 - PHP funkce serialize() nad objektem.

```php
class Test {
   public $public = 1;
   protected $protected = 2;
   private $private = 3;
   private $no = 4;

   /**
    * Výčet názvů proměnných, které se mají serializovat
    * @return array
    */
   public function __sleep() {
      return ['public', 'protected', 'private'];
   }

   public function __wakeup() {
      // ...
   }
}

serialize(new Test());
// výstup -> O:8:"App\Test":3:{s:6:"public";i:1;s:12:"\0*\0protected";i:2;s:17:"
\0App\Test\0private";i:3;}
```

Třetím znakem ve výstupu viz Kód 13 je číslo 8, jedná se o délku řetězce názvy třídy. Z výstupu je dále vidět, že jiné typy viditelnosti atributů se serializují jinak. Privátní proměnné jsou prefixovány \0ClassName\0, protected proměnné znaky \0*\0 a public nejsou prefixovány ničím. Znak \0 je výše zmínění NUL bajt (Pauli, a další, 2013).

Výše zmíněnou nevýhodu metody *__sleep()* lze obejít implementováním rozhraní *Serializable*. Toto rozhraní definuje dvě metody *serialize()* a *unserialize()*. Jestliže třída implementuje toto rozhraní, magické metody *__sleep()* a *__wakeup()* již nejsou volány, ale místo nich jsou volány metody definované v rozhraní. Metoda *serialize()* musí vracet hodnotu *string* serializovaných dat nebo *null*. Ukázka použití viz Kód 14.

Kód 14 - PHP funkce serialize() nad objektem implementujícím \Serializable.

```
class Test2 implements \Serializable {

   public function serialize() {
      return "foobar";
   }

   public function unserialize($str) {
      // ...
   }
}

serialize(new Test2());
// výstup -> C:9:"App\Test2":6:{foobar}
//                           ^-- strlen("foobar")
```

Výsledek této metody je vložen jako součást výsledku funkce *serialize()*. Hlavní struktura serializovaného řetězce zůstává tedy stejná. Tímto způsobem PHP při deserializaci pozná, o jakou třídu se jedná a při deserializaci zavolá příslušnou metodu *unserialize()* na daném objektu. Implementací této metody by mělo být zpětné nastavení příslušných atributů objektu.

Ne všechny interní PHP třídy je možné serializovat. Mnoho z interních tříd buď implementuje rozhraní *Serializable* nebo magické metody *__sleep()* a *__wakeup()*. Pokud ovšem třída nesplňuje tento požadavek, nelze ji spolehlivě serializovat (Achour, 2017). Příkladem může být třída SimpleXMLElement, kterou je možné použít na procházení XML stromů.

3.6.2 Java

Stejně jako PHP i JAVA obsahuje nativní podporu pro serializaci dat. Konkrétně se jedná o balíček *java.io* od verze JDK 1.1. Na rozdíl ale od PHP, kde se jedná o textovou serializaci, JAVA implementuje binární serializaci. Výsledná data jsou zakódována v binární (číslicové) podobě a nejsou tedy lidsky čitelná. Jelikož JAVA je čistě objektový jazyk s výjimkou primitivních datových typů, mluví se především o objektové serializaci. Tedy serializaci celých objektů. K tomu aby mohl být objekt serializován, musí implementovat rozhraní *java.io.Serializable* nebo

java.io.Externalizable. Postačující je, aby rozhraní implementoval předek. Rozraní *Serializable* nedefinuje žádné metody, ale pouze označuje objekt jako připravený k serializaci. Rozhraní *Externalizable* již definuje dvě metody, a sice *writeExternal()* a *readExternal()*. Jak je již z názvu patrné, tyto metody jsou volány při serializaci a deserializaci. Jako parametr metody *writeExternal()* je předán objekt typu *java.io.ObjectOutput*. Obdobně metodě *readExternal()* je předána implementace třídy *java.io.ObjectInput*. Pomocí objektů těchto tříd je možné provést samotnou serializaci. Obě metody nemají žádnou návratovou hodnotu. (Hoff, 2010) Ukázka jednoduchého použití viz. Kód 15.

Kód 15 - Ukázka použití serializace v JAVA.

```
public class Test implements java.io.Serializable {

    private String name;
}

public class Test2 implements java.io.Externalizable {

    private String name;

    public void writeExternal(ObjectOutput out) throws IOException {
       out.writeUTF(name);
    }

    public void readExternal(ObjectInput in)
           throws IOException, ClassNotFoundException {
       name = in.readUTF();
    }
}
```

Při použití rozhraní *Serializable* jsou serializovány i všechny objekty, na které daný objekt drží referenci, s výjimkou statických proměnných, které nejsou serializovány nikdy. Přičemž i tyto objekty musí implementovat rozhraní *Serializable*. Pokud by tomu tak nebylo, je vyhozena výjimka. Tomuto chování se dá zabránit označením proměnné jako *transient* nebo definováním *serialPersistentFields*. Níže následuje ukázka použití viz Kód 16.

Kód 16 - JAVA použití transient a serialPersistentFields

```
public class Test implements java.io.Serializable {

    private transient String secret = "xxxx";
    private String name;
    private static final ObjectStreamField[] serialPersistentFields
            = {new ObjectStreamField("name", String.class)};
}
```

Při použití klíčového slova *transient* nebude daná proměnná nadále serializována. Naopak při definování *serialPersistentFields* jsou uvedeny pouze ty proměnné, které chceme serializovat. Definice *serialPersistentFields* musí být přesně ve výše uvedeném tvaru Kód 16, jinak bude ignorována.

Při použití rozhraní *Externalizable* je vyžadováno definování volného konstruktoru, tedy konstruktoru bez argumentů. Toto omezení není nutné při použití rozhraní *Serializable*.

Dalším rozdílem je, že při implementaci *Serializable* by mělo být definováno tzv. *serialVersionUID*. Pokud definováno není, je generováno z atributů a metod třídy při každé serializaci. Toto id se tedy mění při každé změně třídy. Pokud vygenerované id nesouhlasí s id, které je již serializované, proces deserializace selže, a nebude tak možné objekt obnovit. Z důvodu různých možných implementací výpočtu *serialVersionUID* napříč verzemi JAVY, je doporučeno vždy toto id poskytnout (Garg, 2014). Deklarovaná proměnná musí být označena *static*, *final*, typem *long* s název serialVersionUID nejlépe s *private* viditelností, jak je vidět na ukázce kódu níže Kód 17.

Kód 17 - JAVA příklad serialVersionUID

```
private static final long serialVersionUID = 3762630697919363021L;
```

Samotný proces serializace je pak proveden pomocí třídy *java.io.ObjectOutputStream.* Konkrétně voláním metody *writeObject()*. Třída také poskytuje metody pro serializaci primitivních datových typů *int, boolean*, atd. Konkrétně se jedná o metody definované v rozhraní *java.io.ObjectOutput* jako *writeFloat(), writeInt()* atd., které třída implementuje. Po zavolání jedné nebo více

z uvedených metod se doporučuje zavolat metodu *flush()*, která zajistí, že jakákoliv data, která mohou být stále obsažena v paměti (buffer) budou zapsána do streamu. Na konci celého procesu je ještě vhodné zavolat metodu *close()*, která uvolní všechny reference na zdroje, například na soubor do kterého se bude zapisovat. Při vytváření instance třídy *ObjectOutputStream* je zapotřebí předat do konstruktoru objekt dědící od abstraktní třídy *java.io.OutputStream*. Od této třídy dědí i samotná třída *ObjectOutputStream*, jedná se tedy o návrhový vzor dekorátor. V následující ukázce kódu (Kód 18) jsou serializovaná data zapsána do souboru pomocí *java.io.FileOutputStream*. (Hoff, 2010)

Kód 18 - JAVA serializace pomocí FileOutputStream

(zdroj: https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serial-arch.html)

```
FileOutputStream f = new FileOutputStream("tmp");
ObjectOutput s = new ObjectOutputStream(f);
s.writeObject(new Date());
s.flush();
s.close();
```

Obdobně jako můžeme data serializovat a zapisovat, můžeme také data číst a deserializovat zpět do kódu. K tomu slouží třída *java.io.ObjectInputStream*. Tato třída obsahuje metody pro čtení objektu i priminitvních datových typů. Konkrétně se jedná o metody jako *readObject(), readFloat()* atd.; po ukončení práce s touto třídou je opět vhodné zavolání metody *close()*, která uvolní veškeré reference na zdroje. Podmínkou pro úspěšnou deserializaci je, že definice třídy existuje, že tato třída nebyla změněna od poslední serializace nebo že obsahuje stejné *serialVersionUID*. V opačném případě je vyhozena výjimka. (Hoff, 2010) Ukázka použití viz Kód 19.

Kód 19 - JAVA deserializace pomocí FileInputStream

(zdroj: https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serial-arch.html)

```
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
s.close();
```

Stejně jako u PHP tak i u jazyka JAVA nelze všechny interní třídy spolehlivě serializovat. Dalším omezením je serializace vnitřních tříd. Tento mechanismus není doporučován a nemusí vždy spolehlivě fungovat (Garg, 2014).

3.6.3 JavaScript

Poslední z testovaných jazyků je JavaScript, který stejně jako dva předešlé obsahuje nativní serializaci. V případě JavaScriptu se jedná o formát JSON, který je popsaný v předešlé kapitole JSON.

Jak již bylo řečeno, tento formát vychází z JavaScriptové notace zápisu objektů. Přesto se od této notace v několika věcech odlišuje. Prvním rozdílem je, že u JSON musí být veškeré *string* hodnoty uvozeny ve dvojitých uvozovkách. Klíč musí být vždy typu *string*. Naproti tomu v JavaScript notaci objektů lze zaměňovat jednoduché uvozovky s dvojitými. Klíč může být typu *string* nebo *number* aj. Dalším rozdílem je, že JSON obsahuje jen omezenou sadu datových typů, které rozeznává. Konkrétně se jedná o tento výčet: *String, Number, Object, Array, Boolean, Null*. Oproti tomu JavaScript navíc obsahuje typy jako: *Undefined, Function, RegExps* a také klíčová slova jako *NaN, Infinity, -Infinity*. Použití zápisu, který je možné zhlédnout níže, vyústí v nevalidní JSON, viz Kód 20.

Kód 20 - JavaScript nevalidní JSON.

```
{ "x": undefined };          // nevalidní
{ "key": 'val' };            // nevalidní
```

Z logických důvodů bylo vybráno jen několik výše zmíněných datových typů (klíčových slov). JSON byl navržen jako formát nezávislý na platformě, a proto byly vybrány jen běžné typy, které je možné najít u většiny jazyků.

Funkce, která se využívá pro serializaci dat, se nazývá *JSON.stringify()*. Pro zpětnou deserializaci pak lze využít funkce *JSON.parse()* nebo *eval()*. První zmíněná funkce *stringify()* přijímá tři parametry, přičemž pouze první je povinný. Prvním parametrem jsou samotná data (objekt), která chceme serializovat do JSON. Druhým nepovinným parametrem je funkce, pomocí které můžeme určit, jaké hodnoty zahrnout do JSON nebo pole s výčtem názvu proměnných. (Kramer, 2017) Ukázka použití níže Kód 21.

Kód 21 - JavaScript příklad JSON.stringify()

(zdroj: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify)

```
function replacer(key, value) {

  // Filtering out properties
  if (typeof value === 'string') {
    return undefined;
  }
  return value;
}

var foo = {foundation: 'Mozilla', model: 'box', week: 45, transport: 'car',
month: 7};
JSON.stringify(foo, replacer);
// '{"week":45,"month":7}'
```

Posledním taktéž nepovinným parametrem je možné předat *String* nebo *Number*, kterými se bude výsledný JSON formátovat. Pokud se jedná o číslo, toto číslo určuje počet white space znaků (mezer) použitých ve výsledném formátu. V případě hodnoty *String* bude tato hodnota použita jako white space. Pokud hodnota není předána, žádný znak white space není použit. Viz ukázka Kód 22.

Kód 22 - JavaScript funkce JSON.stringify() s třetím parametrem

(zdroj: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify)

```
JSON.stringify({ uno: 1, dos: 2 }, null, '\t');
// returns the string:
// '{
//     "uno": 1,
//     "dos": 2
// }'
```

Jestliže JavaScriptový objekt obsahuje proměnnou pojmenovanou *toJSON*, která jako hodnotu obsahuje funkci, je při zavolání *stringify()* nad tímto objektem použita právě tato funkce. Viz ukázka Kód 23.

Kód 23 - JavaScript toJSON()

(zdroj: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify)

```
var obj = {
  foo: 'foo',
  toJSON: function() {
    return 'bar';
  }
};
JSON.stringify(obj);        // '"bar"'
JSON.stringify({ x: obj }); // '{"x":"bar"}'
```

Jak již bylo řečeno, pro opačnou konverzi lze využít buď funkce *JSON.parse()*, nebo *eval()*. Obě funkce deserializují JSON řetězec zpět do kódu programu. Funkce *eval()* je obecnější funkce, která parsuje vstup a překládá ho na JavaScriptový kód, který následně vykoná. Z toho důvodu se nedoporučuje používat tuto funkci k deserializaci, jelikož by se mohlo jednat o potenciální bezpečnostní díru v aplikaci, pokud by vstupní JSON obsahoval škodlivý kód. Naproti tomu funkce *JSON.parse()* pouze převádí validní JSON na odpovídající entity JavaScriptu, v opačném případě vyhodí výjimku. Následuje ukázka použití (viz Kód 24):

Kód 24 - JavaScript JSON.parse()

(zdroj: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse)

```
JSON.parse('{}');              // {}
JSON.parse('true');            // true
JSON.parse('"foo"');           // "foo"
JSON.parse('[1, 5, "false"]'); // [1, 5, "false"]
JSON.parse('null');            // null
JSON.parse('alert("hello")');  // Uncaught SyntaxError
```

JSON.parse() ještě přijímá druhý nepovinný parametr. Tímto parametrem může být funkce, která transformuje výsledek. Tomuto callbacku jsou vždy předány dva parametry, a to klíč a hodnota. Viz příklad Kód 25.

Kód 25 - JavaScript JSON.parse() s druhým parametrem

(zdroj: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse)

```
JSON.parse('{"p": 5}', (key, value) =>
  typeof value === 'number'
    ? value * 2 // return value * 2 for numbers
    : value     // return everything else unchanged
);

// { p: 10 }
```

3.7 Ostatní formáty

Samozřejmě ještě existuje celá řada dalších formátů, které lze využít pro serializaci a deserializaci dat. Ne všechny formáty jsou ale implementovány ve všech testovaných jazycích, a také ne všechny formáty jsou určeny pro rychlý přenos dat z jednoho místa na druhé.

3.7.1 YAML

Příkladem formátu, který je spíše určen pro konfiguraci než rychlou serializaci, je YAML. Jedná se textový formát, který je lidsky čitelný i zapisovatelný. Oproti JSON nevyžaduje uvozování do dvojitých uvozovek. Umožňuje psaní komentářů (komentář začíná znakem #). Hierarchie je řešena odsazováním řádků pomocí mezer (2 nebo 4). Podporovány jsou také víceřádkové texty se zachováním

odřádkování. V neposlední řadě je zde i podpora odkazování na různé části dat. (Evans, 2009) Následuje ukázka syntaxe Kód 26.

Kód 26 - YAML syntaxe

(zdroj: http://www.yaml.org/start.html)

```yaml
--- invoice
invoice: 34843
date   : 2001-01-23
bill-to: &id001
    given  : Chris
    family : Dumars
    address:
        lines: |
            458 Walkman Dr.
            Suite #292
        city    : Royal Oak
        state   : MI
        postal  : 48046
ship-to: *id001
product:
    - sku         : BL394D
      quantity    : 4
      description : Basketball
      price       : 450.00
    - sku         : BL4438H
      quantity    : 1
      description : Super Hoop
      price       : 2392.00
tax  : 251.42
total: 4443.52
comments: >
    Late afternoon is best.
    Backup contact is Nancy
    Billsmer @ 338-4338.
```

3.7.2 NEON

Dalším velice podobným formátem je NEON. Jedná se o formát vyvinutý pro konfiguraci Nette frameworku v PHP. Existují ale i implementace do jiných jazyků. Stejně jako YAML i NEON podporuje komentáře, odkazování nebo víceřádkový text,

nevyžaduje uvozovky, hierarchie je řešena odsazováním atd. Hlavním rozdílem oproti YAML je, že NEON podporuje „entities“ (tudíž může být použit pro parsování phpDoc anotací) a umožňuje použít i taby pro hierarchickou strukturu. (Grudl, 2017) Viz ukázku Kód 27.

Kód 27 - NEON syntaxe.

```
# my web application config
php:
    date.timezone: Europe/Prague
    zlib.output_compression: yes  # use gzip
database:
    driver: mysql
    username: root
    password: beruska92
users:
    - Dave
    - Kryten
    - Rimmer
```

3.7.3 Ostatní

Mezi ostatní lze zmínit například Apache Thrift, Colfer, FlatBuffers, BSON a další. Jistě existuje celá řada dalších formátů, které je možné využít pro (de)serializaci dat. Cílem práce ale není otestovat všechny existující formáty, nýbrž porovnat nejpoužívanější z nich.

4 Programovací jazyky

4.1 PHP

Jazyk PHP byl vytvořen dánsko-kanadským programátorem Rasmusem Lerdorfem v roce 1995. Původně se jednalo pouze o skriptovací jazyk, který měl usnadnit vývoj webových aplikací. Od té doby prošel jazyk velkou evolucí a od verze 5, vydané v roce 2004, je objektovým jazykem. I když je PHP dynamickým jazykem, v aktuální verzi 7.1 podporuje typování argumentů metod a jejich návratových hodnot. Na typování proměnných tříd se stále ještě čeká. Tento trend umožňuje PHP kompilátoru lépe analyzovat kód a zapojovat různé optimalizace, které vedou ke zrychlení kódu (Vigh, 2017).

Zajímavostí jazyka je jeho samotný vývoj, neboť za PHP nestojí žádná velká organizace, ale je spravováno skupinou nezávislých vývojářů. Unikátní je způsob přidávání nových věcí do jazyka, veškerá komunikace je totiž vedena na „archaickém" systému mailing list. Jedná se o starší obdobu diskuzních fór s tím rozdílem, že veškerá komunikace probíhá pomocí e-mailů, přičemž tyto diskuze tvoří stromovou strukturu. Pokud někdo přijde s novým nápadem, je třeba napsat RFC (Request For Comments) dokument, který musí splňovat určité náležitosti jako odpovědi na otázky co, proč, jak atd. O tomto RFC se dále hlasuje a teprve poté je buď zařazeno do PHP, nebo ne.

Důležitým rokem pro PHP byl rok 2012, kdy byl poprvé představen nástroj Composer. Jedná se o utilitu, která umožnuje spravovat závislosti PHP aplikace na ostatních knihovnách. Composer řeší stahování knihoven a jejich vzájemnou závislost. Lze definovat požadované verze, stabilitu knihoven (alfa, beta…), závislosti pro vývojovou verzi projektu a další. Composer lze provázat i s GIT[11] repositáři, přičemž označení kódu v GIT novou verzí automaticky vytvoří záznam o

[11] GIT – jeden z nejrozšířenějších verzovacích systém pro usnadnění vývoje aplikací.

nové verzi v Composeru. Veškerá konfigurace je zapsána v JSON formátu v souboru *composer.json*.

Dnes PHP patří mezi jeden z nejpoužívanějších a nejoblíbenějších jazyků k tvorbě webových aplikací (Dvořáková, 2016). Některé velké projekty jako je Wikipedie nebo Facebook byly naprogramovány v PHP. PHP není jen webovou technologií, ale lze v něm psát konzolové aplikace, nebo dokonce i desktopové aplikace.

4.2 JAVA

Jazyk JAVA byl obdobně jako PHP poprvé představen v roce 1995. Na rozdíl od PHP za JAVOU stála firma Sun Microsystems, která později přešla pod Oracle. JAVA je robustní objektově orientovaný jazyk aktuálně ve verzi 8. Tato verze přinesla několik novinek, například proudové zpracování dat, lambda výrazy aj. S JAVOU je možné programovat desktopové, konzolové, webové, mobilní i velké enterprise aplikace. JAVA aplikace lze spustit takřka na kterémkoliv operačním systému, jelikož samotný kód aplikace je na platformě nezávislý. To ovšem neplatí pro JVM (Java Virtual Machine), který pomocí virtualizace interpretuje bajt kód aplikace a spouští odpovídající nativní instrukce konkrétního procesoru a konkrétní systémová volání operačního systému. Tento virtuální stroj je již závislý na platformě, nicméně je napsán pro širokou škálu operačních systémů (Windows, Linux nebo OS X).

Obdobně jako pro PHP a jiné programovací jazyky, tak i pro JAVU existuje systém pro správu závislostí na knihovnách třetích stran. V JAVĚ tyto systémy navíc definují sestavovací proces (build) aplikace. Mezi nejstarší používané programy v této oblasti patří Apache Ant. Ant nedefinuje žádné externí repositáře, ze kterých by se mohli stahovat knihovny, slouží pouze pro sestavování aplikace pomocí konfigurace zapsané v *build.xml* souboru. Pro doplnění této funkčnosti vznikl Apache Ivy, který řeší stahování knihoven z repositářů. Dalším systémem je Apache Maven. Tento systém již definuje externí repositáře, ze kterých je možné automaticky stahovat závislosti a také popisuje sestavovací proces. Konfigurace je opět zapsána ve formátu XML v souboru *pom.xml*. Třetím nejpoužívanějším systémem je Gradle, který staví na zkušenostech z Antu a Mavenu. Konfigurace v něm není zapsána pomocí XML, ale ve vlastním formátu *build.gradle*, který je méně „upovídaný“ než

XML. Gradle je obecně nástroj na zautomatizování věcí, neslouží pouze k sestavování aplikací, ale lze jej využít například na automatické generování a odesílání měsíčních výkazů (Kotačka, 2013).

Obecně okolo ekosystému JAVY vzniklo a vzniká celá řada frameworků a jiných podpůrných nástrojů. Zřejmě i proto patří JAVA podle TIOBE Index již několik let na první místo v oblíbenosti programovacích jazyků (Jansen, 2017).

4.3 JavaScript

Posledním použitým jazykem je JavaScript (zkráceně JS). Ačkoliv název technologie obsahuje slovo Java, s tímto jazykem nemá příliš společného. Sdílí snad jen to, že vznikly ve stejném roce 1995. Autorem jazyka je Brendan Eich. Primárně jazyk cílil na mezeru mezi HTML a klasickými serverovými jazyky. Jeho snahou bylo rozhýbat webové stránky, přidat živou dynamičnost, a umožnit tak lepší zážitek při surfování internetem. Jazyk byl v roce 1997 standardizován asociací ECMA (European Computer Manufacturers Association) a dále se vyvíjí jako jedna s implementací ECMAScriptu vedle například ActionScriptu. Od verze ECMA 6 (aktuální verze) jazyk podporuje konstrukce známé z jiných objektově orientovaných jazyků, a sice zápis třídy, dědění tříd a jiné syntaktické konstrukce. I v předchozích verzích bylo možné toto chování imitovat, ale zapisovalo se pomocí jiné syntaxe (Kramer, 2017).

Na rozdíl od jazyků JAVY nebo PHP byl JS primárně určen pro zpracování na klientské straně, konkrétně ve webovém prohlížeči uživatele. Což vede k řadě problémů, s možnou bezpečností, rozdílným chováním napříč běhovými prostředími nebo také s vývojem nových verzí, protože je třeba často čekat, než danou novinku budou podporovat všechny hlavní verze prohlížečů. Velkou změnu v této oblasti přinesla technologie NodeJS. Nejedná se však o jedinou, ale ani první technologii, která umožňuje spouštět JS na straně serveru. Právě možnost spouštění JS na straně serveru umožnila bouřlivý vývoj různých frameworků a tvorbu tzv. izomorfních aplikací. Jedná se o aplikace, které jsou napsány v jednom programovacím jazyku (JS), a to na straně serveru i klienta..

Stejně jako pro jiné jazyky i pro JS existuje správce závislostí na externích knihovnách. Nejpoužívanějším nástrojem je NPM. Tento systém obdobně jako

Composer zapisuje konfiguraci do JSON souboru, konkrétně *package.json*. Aktuálně se jedná o největší repositář knihoven na světě, který obsahuje přes 350 000 balíčků, což je dvakrát více než druhý Apache Maven (Brown, 2017).

5 Metodika testování

5.1 Existující benchmarky

Prvním krokem k vytvoření benchmarku je podívat se na již existující řešení a zhodnotit závěry, ke kterým jejich autoři došli. Nejvíce benchmarků lze najít na téma XML vs. JSON jako například „Comparison of JSON and XML Data Interchange Formats: A Case Study“ (Nurseitov, a další, 2009), „JSON vs. XML: Some hard numbers about verbosity“ (Pragmateek, 2013) nebo „Improving Data Transmission in Web Applications via the Translation between XML and JSON“ (Wang, 2011). Srovnání s ostatními formáty již není tolik frekventované. Velká většina testování formátů je napsaná v jazyku JAVA, což lze vysvětlit velkou oblíbeností jazyka, viz kapitolu Java. Výrazně méně testování formátů je napsáno v jazycích PHP nebo JavaScript.

Jedním z nalezených benchmarků, který se zabývá testováním JSON formátu v JAVĚ je „Benchmark of Java JSON libraries“ (Renaud, 2017). Tento benchmark srovnává více než patnáct knihoven. Výsledky jsou přehledně prezentovány ve formě grafů i tabulek. Nechybí uvedení verze knihoven ani hardwarová a softwarová výbava počítače, kde byl benchmark spuštěn. Dalším, již více robustnějším, benchmarkem je „Benchmark comparing serialization libraries on the JVM“ (Saloranta, 2016). Tento benchmark testuje mnoho formátů. Výsledky jsou zobrazeny opět ve formě grafů i tabulek. Nicméně chybí popis verzí knihoven a testovacích dat. Dále je možné zmínit článek „Comparative Survey of Object Serialization Techniques and the Programming Supports“ (Maeda, 2011). V této práci je testováno šest formátů z hlediska jejich paměťové náročnosti. Chybí ale popis verzí, hardware a software počítače a celkově testování rychlosti serializace a deserializace. Mezi další lze zmínit „Serialization Benchmark“ (Shabanov, 2014) nebo již citovaný „Comparison of JSON and XML Data Interchange Formats: A Case Study“ (Nurseitov, a další, 2009).

V oblasti PHP je možné zmínit „Benchmarking BSON, JSON, and Native Serializing in PHP“ (Suarez, 2016). Tento benchmark testuje tři nativní funkce PHP, a sice *serialize(), json_encode(), bson_encode()*. Výsledky jsou prezentovány pouze číselně,

bez grafů. Opět ale chybí uvedení verze, v tomto případě verze PHP, chybí také popis hardwaru a testovacích dat. Dalším benchmarkem je „PHP Serialization Benchmarks“ (Sági-Kazár, 2017), který testuje tři formáty (JSON, nativní formát a MsgPack). Výsledky a nedostatky jsou totožně s předchozím zmíněným benchmarkem.

Poslední technologií je JavaScript. Zde lze jmenovat „Benchmarks“ (Monsch, 2016), který testuje pět formátů (Avro, MsgPack, JSON, Protobuf, PSON) v běhovém prostředí NodeJS. Výsledky jsou zobrazeny formou grafů. Přidán je i popis hardwaru hostitelského počítače. V benchmarku je ale zmíněna pouze verze Avro implementace, nikoliv verze ostatních knihoven.

Najít lze i další benchmarky pro zmíněné jazyky. Většina z nich ale již není aktuální nebo neobsahuje dostatečně relevantní informace k zhodnocení výsledků. Žádný z nalezených benchmarků nesrovnává výsledky napříč programovacími jazyky. Často chybí uvedení verzí knihoven a hardwaru počítače, na kterém byl test spuštěn, nebo popis a velikost testovacích dat (častý problém u výše zmíněných). Většina benchmarků také obsahuje jen malý počet formátů nebo malý počet knihoven implementujících danou funkčnost. Právě z těchto důvodů byl vytvořen vlastní benchmark.

5.2 Princip měření

Z hlediska určení výsledků benchmarku můžeme měřit několik veličin. Jednou z nich je čas potřebný k dokončení serializace a čas potřebný k dokončení deserializace dat. Dále je možné měřit počet operací za sekundu, tuto veličinu používá například „Benchmark of Java JSON libraries“ (Renaud, 2017). U serializovaných dat se dále můžeme zaměřit na velikost těchto dat.

Pro měření byl nakonec vybrán čas serializace a deserializace. Tento údaj je jednoduše interpretovatelný a srozumitelný. Tedy čím menší číslo, tím rychlejší knihovna – formát. Jako jednotky jsou použity milisekundy. Do výsledků je dále zahrnuta velikost serializovaných dat, údaj, který je jednoduše interpretovatelný a srozumitelný. Čím menší číslo, tím lepší hodnocený výsledek. Jako jednotky jsou použity kilobyty (kB). Ukazatel operace za sekundu (ops/sec) není do výsledku

zahrnut. Tento údaj v podstatě říká, kolikrát byl daný proces spuštěn v jedné vteřině. Tato veličina ale není příliš intuitivní. Když získáme čas v známých jednotkách jako milisekundy, dokážeme si tento údaj lépe představit. Pokud ale dostaneme výsledek v jednotkách ops/sec, je to číslo, které nic moc neříká. Pokud se výsledky různých knihoven budou lišit jen o několik jednotek, představuje to významný rozdíl nebo ne? Navíc pro tuto hodnotu platí čím větší číslo, tím lepší výsledek. Z tohoto důvodu je ve výsledcích měřen jen čas a velikost serializovaných dat stejně jako v „Benchmark comparing serialization libraries on the JVM“ (Saloranta, 2016) nebo „Benchmarks“ (Monsch, 2016) a dalších.

Měření času jednoho průchodu trvá řádově pouze několik málo mikrosekund, samozřejmě záleží na velikosti vstupních dat. Proto by mohlo docházet k velkému rozptylu naměřených hodnot. Z tohoto důvodu je vhodnější provést vícero měření. Stejná data jsou cyklicky serializována/deserializována a je změřen celkový čas; celý tento proces je navíc ještě několikrát zopakován. Kód 28 ukazuje pseudo implementaci tohoto řešení v JAVĚ.

Kód 28 - Způsob měření času

```
for (int j = 0; j < OUTER; j++){
    start = System.nanoTime();
    for (int i = 0; i < INNER; i++) {
        // zavolej (de)serialize metodu
        //...
    }
    time = System.nanoTime() - start;
    // zpracuj (ulož) výsledný čas
    //...
}
```

Samotné naměřené hodnoty posuzované individuálně nejsou až tak významné. Důležité je jejich porovnání mezi sebou, jelikož výsledné hodnoty, naměřené pomocí spuštění benchmarku, se budou lišit na každém počítači, a to, ať už z důvodu rozdílného hardwaru nebo operačního systému, popřípadě jeho aktuální vytíženosti. Nicméně vzájemné porovnání by mělo zůstat zachováno.

5.3 Testovací data

Důležitou součástí testování serializace a deserializace je výběr testovacích dat. Testovací data by měla být dostatečně veliká a rozmanitá. Rozmanitost je myšlena z hlediska datových typů, tedy kombinovat čísla, desetinná čísla, logické hodnoty, řetězce, pole atd. Pro benchmark byla nakonec vybrána náhodně generovaná data ze služby http://www.json-generator.com/. Jedná se o testovací data popisující fiktivní uživatele. Ukázka dat v JSON formátu viz Kód 29.

Kód 29 - Testovací data

```
{
    "persons": [
        {
            "id": "58344fae916de7ac6470585b",
            "index": 0,
            "guid": "739fc0df-2705-4b8f-b4d7-b74b2c68bf63",
            "isActive": false,
            "balance": "$2,408.21",
            "picture": "http://placehold.it/32x32",
            "age": 21,
            "eyeColor": "brown",
            "name": "Harper Cline",
            "gender": "male",
            "company": "TOYLETRY",
            "email": "harpercline@toyletry.com",
            "phone": "+1 (888) 549-2594",
            "address": "292 Lawrence Avenue, Freetown, Delaware, 4951",
            "about": "Ex nisi veniam deserunt est aliqua do aliqua excepteur qui
amet consequat laboris nulla. Sunt Lorem in dolor eu sit occaecat labore irure
excepteur laborum exercitation laborum do tempor. Non laboris sunt consectetur
enim et cupidatat irure commodo consequat sit est. Lorem ut minim sint occaecat
quis fugiat Lorem enim. Ut Lorem velit Lorem ex. Velit ad proident mollit irure
occaecat eu fugiat in laboris eiusmod aliquip ut.\r\n",
            "registered": "2016-11-01T11:32:45 -01:00",
            "latitude": 3.262226,
            "longitude": 10.952296,
            "tags": ["reprehenderit", "pariatur", "aliqua", "dolore", "tempor"],
            "friends": [
                {"id": 0, "name": "Byrd Booth"},
                {"id": 1, "name": "Baxter Mejia"},
                {"id": 2,"name": "Jana Daugherty"}
            ],
            "greeting": "Hello, Harper Cline! You have 6 unread messages.",
            "favoriteFruit": "strawberry"
        },
        …
    ]
}
```

Data splňují požadavek na rozmanitost, obsahují čísla, pole, řetězce různých délek, kolekci objektů atd. Celkem je pro benchmark použito deset uživatelských záznamů. Velikost výsledných dat ve formátu JSON je přibližně 12 kB (kilobytů). Tato data jsou

stejná pro všechny tři programovací jazyky. Obdobnou strukturu dat používají i „Benchmark of Java JSON libraries“ (Renaud, 2017) nebo „Benchmark comparing serialization libraries on the JVM“ (Saloranta, 2016) pro svoje benchmarky.

6 Implementace

Vytvořený benchmark je naprogramován ve třech programovacích jazycích, a sice JAVA, PHP a JavaScript. Jednotlivé aplikace jsou naprogramovány jako konzolové programy, neobsahují GUI (grafické uživatelské rozhraní). Spustit je lze takřka na libovolném operačním systému jako Linux, Windows aj. Jedinou podmínkou je, aby operační systém umožňoval spouštět aplikace napsané právě ve výše zmíněných jazycích. Nicméně pro automatické spuštění všech benchmarků je napsán skript pro operační systém Linux.

Při spuštění benchmarku je možné nastavit několik parametrů. Tyto možnosti jsou stejné pro všechny tři aplikace. Jmenovitě se jedná o:

- *-o, --outer <n>* → vnější počet opakování (viz Kód 28).
- *-i, --inner <n>* → vnitřní počet opakování (viz Kód 28).
- *-t, --data <s>* → cesta k testovacím datům.
- *-r, --result <csv|console>* → formát výstupu.
- *-f, --format <native|json|xml|protobuf|avro|msgpack>* → testuj pouze zadaný formát.
- *-d, --out-dir <s>* → cesta k výstupní složce (pokud je formát výstupu nastaven na csv).
- *-c, --chatty* → při testování vypisuje aktuální činnost.

Všechny tyto volby lze kombinovat, například tedy spuštění benchmarku s možnostmi, viz Kód 30:

Kód 30 - Možnosti spuštění benchmarku (dlouhá verze)

```
--inner 10 --outer 10 --data ./testdata.json --result csv --format json --out-dir ./ --chatty
```

Tyto možnosti spustí konkrétní benchmark s deseti vnitřními a deseti vnějšími opakováními, testovat se budou pouze JSON knihovny. Výstup chceme ve formě csv (viz níže) uložit do aktuální složky, cesta k testovacím datům je v aktuálním adresáři

pod jménem *testdata.json,* dále chceme podrobný výpis testování. Příklad zápisu možností spuštění Kód 30 je ekvivalentní (viz Kód 31):

Kód 31 - Možnosti spuštění benchmarku (krátká verze)

```
-i 10 -o 10 -t ./testdata.json -r csv -f json -d ./ -c
```

Tím se zápis podstatně zkracuje.

V případě benchmarku pro PHP (obdobně pro ostatní jazyky) je výstupem spuštění jednotlivé aplikace s možností -r csv, několik souborů, které poskytují informace o výsledku testu. Celkem se vytvářejí čtyři soubory pro každý jednotlivý benchmark. Struktura těchto souborů je stejná pro všechny tři technologie. Prvním ze souborů je textový soubor *technologie*-info.txt (například php-info.txt). Tento soubor obsahuje informace o provedeném testu jako datum spuštění (ISO 8601 formát), počet vnitřních a vnějších opakování, velikost testovacích dat a verzi prostředí, ve kterém byl test spuštěn. Ukázka souboru – viz Kód 32.

Kód 32 - Výstupní soubor php-info.txt

```
PHP version: 7.1.4
Test data size (raw): 13.08 kB
Outer repetition: 100
Inner repetition: 100
Date: 2017-06-05T12:28:16+00:00
```

Dalším z výstupních souborů jsou soubory *technologie*-serialize.csv a *technologie*-deserialize.csv. CSV soubory jsou textové soubory, kde jsou jednotlivé ucelené záznamy odděleny znakem enter (nový řádek). Oddělení jednotlivých hodnot v řádku je pomocí speciálního znaku (nejčastěji čárka, středník, tabulátor atd.), v našem případě se jedná o středník (;). Tento formát ještě umožňuje definovat hlavičku dat na prvním řádku souboru, oddělení je opět pomocí zvoleného znaku. Tyto soubory je možné otevřít například v tabulkových programech jako MS Excel a dalších. Dva výše zmíněné soubory obsahují data o serializaci nebo deserializaci, na prvním řádku (jako hlavička) jsou vypsány všechny testované knihovny, název je složen z „název technologie – název formátu – název knihovny verze“. Na dalších řádcích následují jednotlivé naměřené časové hodnoty. Veškeré časy jsou v milisekundách a oddělovač desetinných míst je tečka. Výsledný počet řádků je dán

počtem vnějšího opakování plus jedna (hlavička). Ukázka výstupu PHP serializace pro deset „opakování (viz Kód 33):“

Kód 33 - Výstupní soubor php-serialize.csv

```
"php - avro - apache/avro 1.8.1";"php - json - PHP native json";…;…;…;…;…
1.013;0.062;0.1001;0.0801;0.0319;0.2148;0.2239;0.025;3.0279;0.9069;0.8719;0.9091
1.0669;0.057;0.0739;0.078;0.015;0.1998;0.2511;0.0238;2.9249;0.9232;0.8631;1.049
1.122;0.056;0.0789;0.0792;0.015;0.2091;0.2029;0.0231;3.3069;0.8469;0.8891;1.0421
1.158;0.0679;0.0639;0.0908;0.015;0.237;0.216;0.0229;3.2139;0.8171;1.2081;1.025
0.983;0.0591;0.0629;0.0572;0.0138;0.2451;0.2232;0.0231;3.0332;1.0571;0.953;0.901
1.087;0.071;0.0629;0.056;0.0138;0.2511;0.2019;0.0219;3.6058;0.9229;1.127;0.998
1.0269;0.066;0.0639;0.0548;0.0222;0.2789;0.2141;0.0231;2.964;0.906;1.0691;0.8981
1.0791;0.0551;0.0629;0.0548;0.0138;0.2301;0.2348;0.0229;3.1312;0.772;1.1981;0.9599
1.229;0.0548;0.0632;0.0551;0.0141;0.2139;0.1969;0.0219;3.587;0.771;1.101;1.049
1.4901;0.0551;0.0629;0.0551;0.0141;0.221;0.2019;0.0219;3.0251;0.772;0.9992;0.9251
```

Posledním z vygenerovaných souborů je soubor *technologie*-summarize.csv. Na prvním řádku souboru je hlavička s nadpisy sloupců. V prvním sloupci jsou názvy knihoven, formát jmen je složen stejně jako v předešlých souborech. V druhém sloupci jsou spočítány průměrné časy serializace v milisekundách pro každou knihovnu zvlášť, jedná se o aritmetický průměr nad všemi vnějšími opakováními. Obdobně ve třetím jsou průměrné časy deserializace. V posledním sloupci se pak nachází výsledná velikost serializovaných dat v kB (kilobytech). Ukázka podoby souboru pro PHP (viz Kód 34):

Kód 34 - Výstupní soubor php-summarize.csv

```
Name;"Serialize (ms)";"Deserialize (ms)";"Size (kB)"
"php - avro - apache/avro 1.8.1";1.0945;1.7282;8.7461
"php - json - PHP native json";0.0576;0.0916;10.8506
"php - json - bukka/php-jsond 1.3.0";0.064;0.1024;10.8506
"php - json - salsify/json-streaming-parser 6.0.1";0;5.4638;0
"php - json - seld/jsonlint 1.5";0;12.8439;0
"php - json - webmozart/json 1.2.2";0.057;0.0956;10.8506
"php - msgpack - msgpack/msgpack-php 2.0.1";0.0147;0.0325;9.6484
"php - msgpack - onlinecity/msgpack-php";0.2149;0.9006;9.6797
…
```

Další možností spuštění je vypsání výsledků přímo do konzole, možnost *-r console* (jedná se o defaultní chování). Způsob prezentace výstupu je formou tabulek, které

jsou sestaveny pomocí znaků jako +, -, | a dalších. Při použití této volby se nevytvářejí žádné soubory. Kompletní výsledek je vytisknut přímo do konzole. Ukázka výstupu PHP benchmarku viz Kód 35.

Kód 35 – Výstup PHP benchmarku do konzole

```
...
---------------------------------------- -------------------- ------------------ -----------------
 Name                                     Serialize (ms)       Deserialize (ms)   Size (kB)
---------------------------------------- -------------------- ------------------ -----------------
 php - native - PHP serialize             0.3732               0.623              13.9668
---------------------------------------- -------------------- ------------------ -----------------
...
```

Aby byl proces spuštění a získávání výsledků více automatizovaný, jsou součástí benchmarků ještě dva podpůrné skripty napsané pro systém Linux. První z nich usnadňuje nastavení prostředí pro spuštění testů pomocí technologie Docker. Druhý pak umožňuje hromadné spuštění benchmarků a navíc generuje dodatečné soubory, které jsou spojením výsledků benchmarků. Z těchto dat dále generuje grafy, které umožňují rychlé zhodnocení výsledků. Tyto skripty nejsou nutnou součástí benchmarků, ale výrazně zjednodušují práci. Více na toto téma je v kapitolách níže (Docker, Hromadné spuštění testů).

6.1 Implementace v PHP

Pro naprogramování benchmarku pro jazyk PHP byly použity objektové vlastnosti jazyka. Konkrétně implementace míří na kompatibilitu s verzí PHP 5.6 nebo vyšší. Pro správu závislostí podpůrných knihoven a serializačních knihoven je použit nástroj Composer (více v kapitole o PHP). Veškeré třídy jsou logicky seskupeny v namespace s název *Benchmark.*

Jelikož PHP automaticky nepodporuje načítání tříd ze souborů, je pro tento účel použita knihovna *nette/robot-loader.* Tato knihovna projde veškeré předané složky a načte z nich třídy, které je dále možné používat bez nutnosti volání funkcí *include* nebo *require*. Dále z nalezených tříd generuje cache pro rychlejší načítání, není nutné pokaždé procházet soubory. Tato knihovna výrazně usnadňuje práci při vývoji PHP aplikací. Další použitou knihovnou je *symfony/console*. Tato knihovna

umožňuje definici a následné parsování spouštěcích parametrů a jejich argumentů, například parametry pro počet opakování, výběr testovaného formátu a další. Dále umožňuje formátování výstupů do konzole včetně vypsání znakových tabulek, nadpisů aj. Posledními dvěma knihovnami jsou *nette/utils* a *league/csv*. První zmíněná knihovna obsahuje některé funkce, které usnadňují práci například s formátováním řetězců, čísel, převodů jednotek atd. Druhá zmíněná knihovna usnadňuje práci s vytvářením CSV souborů, které jsou použity pro ukládání výsledků testů. Zbylé závislosti definované v souboru *composer.json* jsou použity pro samotné testování serializace a deserializace.

Některé testované knihovny nejsou definovány pomocí Composer. Jedná se buď o knihovny, které jsou naprogramovány v jazyku C jako rozšíření PHP, nebo zkrátka jen nepodporují Composer, tyto knihovny jsou umístěny ve složce *libs*. Výhodou C rozšíření do PHP je, že takový kód bude teoreticky vždy rychlejší než ekvivalentní kód napsaný v PHP. Problémem ale je, že na webhostingových serverech často nemáme oprávnění přidávat vlastní rozšíření.

Vstupním bodem do aplikace je soubor *init.php*, který se stará o inicializaci funkcí knihoven jako je zmíněné načítání tříd nebo definování parametrů spouštění. Dále je řízení předáno třídě *RunCommand*. V této třídě je zapsaná samotná konfigurace spouštěcích parametrů, jejich parsování a výchozí hodnoty. Po úspěšném zkontrolování vstupních parametrů je vytvořen objekt *Config,* který obsahuje nastavení benchmarku. Poté se přejde ke spuštění měření. V objektu *Config* je mimo počet opakování a testovacích dat i seznam všech tříd, které implementují rozhraní *IMetric*. Každá třída implementující toto rozhraní bude zahrnuta do testování. Toto rozhraní definuje dvě metody a to *run()* a *getInfo()*. První metoda by měla vracet instanci třídy *MetricResult* a implementovat měření metod serializace a deserializace. Druhá zmíněná metoda by měla vracet instanci třídy *Info*, která obsahuje informace o názvu knihovny, verzi, příslušnosti k formátu a webovou adresu projektu. Pro snadnější přidávání knihoven do testování je zde připravena abstraktní třída *AMetric*, která implementuje rozhraní *IMetric.* Tato třída implementuje metodu *run()* a dále přidává několik metod, které jsou postupně volány. Jedná se o metody *prepareBenchmark()* – volána jednou na začátku,

prepareDataForSerialize() – volána jednou před serializací, *serialize()* – abstraktní metoda pro serializaci dat, *prepareDataForDeserialize()* – volána jednou před deserializací a *deserialize()* – abstraktní metoda pro deserializaci dat.

Měření časů podle Kód 28 je prováděno pomocí funkce *microtime()*, která vrací aktuální Unixové časové razítko v sekundách s přesností na mikrosekundy. Data jsou serializována do datového typu *string* a měření velikosti serializovaných dat je poté prováděno pomocí funkce *strlen()*.

Po dokončení měření všech knihoven se výsledky nacházejí v poli objektů *MetricResult.* Podle zvoleného formátu výstupu dat jsou data buď prezentována ve formě CSV souborů (třída *CsvOutput*), nebo jsou zobrazena přímo do konzole (třída *ConsoleOutput*).

6.2 Implementace v Java

Bechmark naprogramovaný v Jazyce JAVA se snaží dodržet podobnou logickou strukturu tříd jako v PHP. Implementace benchmarku míří na verzi JDK 1.8 a novější. Pro správu závislostí a buildování aplikace je použit MAVEN (více v kapitole JAVA). Veškeré třídy jsou logicky seskupeny do balíčku (package) *benchmark.java*.

Pro nastavení parametrů spuštění a následné parsování je použita knihovna *commons-cli*. Jedná se o knihovnu, která spadá pod organizaci Apache Commons. Tato knihovna značně usnadňuje práci s parametry předanými při spouštění programu. Další knihovnou je *vdmeer/asciitable*, jedná se knihovnu, která umožňuje vytvářet znakové tabulky, které jsou použity pro prezentaci výsledků v konzoli v případě použití možností *-r console*. Poslední knihovou je *ronmamo/reflections*. Tato knihovna mimo jiné poskytuje funkčnost pro hledání tříd implementujících určité rozhraní, v našem případě *IMatric*. Veškeré ostatní závislosti uvedené v souboru *pom.xml* jsou testované knihovny.

Jak již bylo řečeno, struktura tříd je obdobná jako v případě PHP. Hlavní spouštěcí třídou je třída *Init*. V této třídě je nastavení a parsování parametrů a jejich argumentů. Tato třída dále vytváří objekt *Config*, který obsahuje nastavení benchmarku a také seznam všech tříd implementující rozhraní *IMetric*. Pro jednodušší použití je připravena abstraktní třída *AMetric*. Poté se již přejde

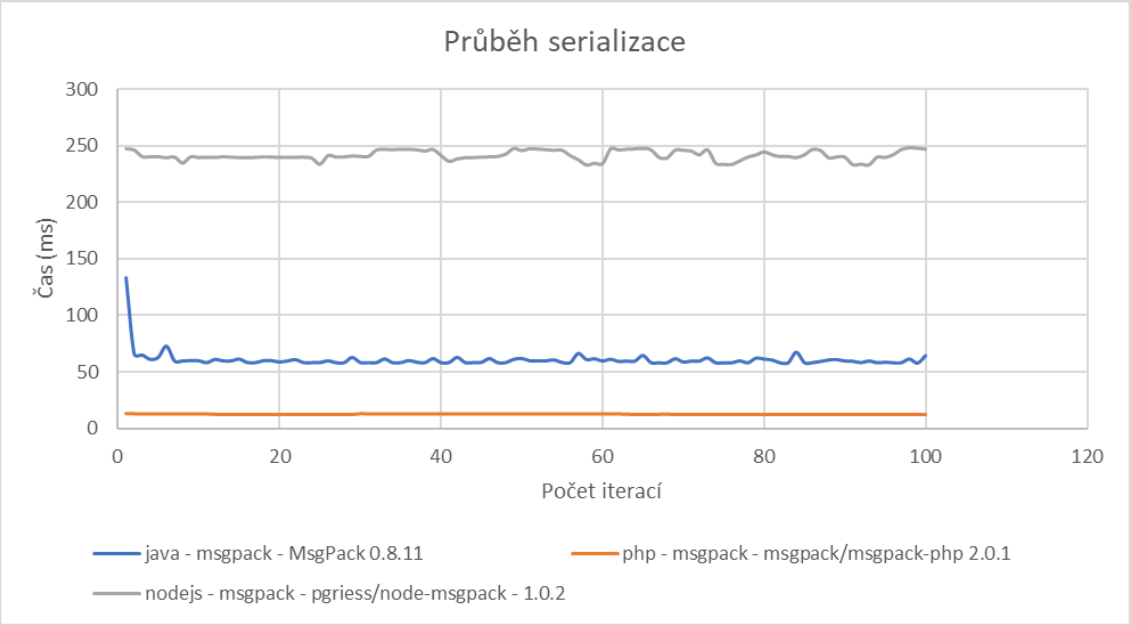
k samotnému měření. Po skončení měření jsou výsledky v seznamu obsahující objekty *MetricResult*. Podle zvoleného formátu výstupu jsou výsledky dále transformovány buď do CSV souborů, nebo zobrazeny do konzole ve formě tabulek.

Měření je opět implementováno podle Kód 28. Čas je konkrétně odečítán pomocí metody *System.nanoTime()*. Tato metoda vrací nejpřesnější dostupný časovač systému v nanosekundách. Při serializaci dat v JAVĚ nám většina knihoven nabízí více možností, do jakých struktur data serializovat. Naprostá většina knihoven podporuje serializaci do objektů typu *java.io.OutputStream*, popřípadě *java.lang.String*. Pro serializaci byl vybrán potomek třídy *java.io.OutputStream*, konkrétně *java.io.ByteArrayOutputStream*. Výhodou této konkrétní třídy je možnost získání pole bytů, které je možné použít k vytvoření instance třídy *java.io.ByteArrayInputStream,* ze které je prováděna deserializace. Při vícenásobné serializaci stejných dat je možné *ByteArrayOutputStream* použít několika způsoby. Prvním způsobem je při každé nové serializaci vytvářet novou instanci objektu, druhým je zapisovat data stále do stejného objektu. V neposlední řadě lze využít metodu *reset()*, která vymaže data ze streamu. V benchmarku je použita první zmíněná metoda a sice vytváření nové instance objektu při každém novém průchodu. Použití ostatních zmíněných metod nemá vliv na výsledek testů. Obdobný proces používají i (Renaud, 2017) nebo (Saloranta, 2017). Zjištění velikosti serializovaných dat je pouze záležitostí zavolání metody *size()* na objektu *ByteArrayOutputStream*, která vrací aktuální počet bytů.

Pro JAVU již existuje hotový framework, který je určený pro měření rychlosti kódu. Jedná se o Java Microbenchmark Harness zkráceně JMH. Jak již název napovídá, jedná se o framework, který měří v přesnosti mikrosekund. Metody, které chceme měřit, stačí opatřit anotací a o zbytek se postará JMH. Další výhodou je, že framework za nás řeší několik věcí, na které je třeba si dát u JAVY pozor, zejména pak na její optimalizace. Ty mohou zkreslit výsledek, zejména pokud se pohybuje v mikrosekundové přesnosti. Důvody, proč tento framework není použit, jsou zejména v rozdílnosti formátu výsledků. Pro další zpracování výstupních hodnot je potřeba zachovat konzistenci mezi PHP a JavaScriptem, viz kapitolu Hromadné spuštění testů. Dále také nevyžadujeme přesnost na mikrosekundy, ale stačí nám

milisekundová přesnost. Proto není potřeba se tak detailně zabývat všemi možnými optimalizacemi. Ignorovat je by ale byla také chyba.

Veškerá testovací data jsou načítána za běhu aplikace ze souboru a až poté předávána do metody. Tím je zabráněno optimalizaci, která vypočítává statický kód během kompilace. Další optimalizací je tzv. *dead-code elimination*. Jedná se proces, kdy metoda, se kterou není dále pracováno, nemá návratovou hodnotu nebo neovlivňuje jiné objekty, se vůbec neprovede. Proto mají metody *serialize()* a *deserialize()* návratové hodnoty a s objekty *OutpuStream* a *InpuStream* se nadále pracuje, aby k této situaci nedošlo. Dalším termínem je warm-up (zahřátí). Definice podle (Shipilev, 2013) zní „*waiting for the transient responses to settle down*". Na Graf 1 jsou zachyceny časy serializace tří knihoven ve třech jazycích. Detailnější popis hardwaru a softwaru pro tuto chvíli není podstatný.



Graf 1 - Průběh serializace dat

(zdroj: autor)

Na grafu Graf 1 je vidět, že u JAVY na rozdíl od PHP a JavaScriptu, je počáteční skok, který po několika málo iteracích klesá. Postupně časy konvergují k určité hodnotě, která je v tomto případě kolem 60 ms. Tedy zhruba po deseti iteracích můžeme prohlásit, že jsme dosáhli stavu *warm-up*. U JavaScriptu můžeme také pozorovat určité kmitání, ale není tam tak výrazný skok na začátku jako v případě JAVY. Jazyk PHP má pak v tomto případě takřka hladký průběh. Obdobný průběh je možné

pozorovat i u ostatních knihoven, kdy je bez výjimky vidět počáteční skok u průběhu JAVY. Pokud vezmeme medián těchto hodnot nebo aritmetický průměr, pak po dostatečném množství iterací, jsou tyto skokové extrémní hodnoty eliminovány.

V JAVĚ existuje ještě celá řada dalších optimalizací. Benchmark je věčným bojem proti optimalizacím (Shipilev, 2013). Jestliže se ale budeme zabývat všemi možnými optimalizacemi, můžeme dojít do stavu, kdy samotný benchmark bude měřit nějaké výsledky, ale s reálnými časy v aplikaci nebude mít mnoho společného, jelikož se mohou projevit právě optimalizace.

6.3 Implementace v JavaScriptu

Benchmark v JavaScriptu (JS) je napsán pro kompatibilitu s ECMAScript 5. Jako běhové prostředí je použit NodeJS s kompatibilitou pro verzi 6 a více. Pro řízení závislostí je použit NPM. Jelikož JS je objektový jazyk, ale není třídní (neobsahuje třídy), je implementace benchmarku lehce odlišná od PHP nebo JAVY.

Pro nastavení a následné parsování vstupních argumentů je použita knihovna *commander*. Nastavení argumentů je stejné jak v případě JAVY nebo PHP. Další použitou knihovnou je *cli-tables*. Jak název napovídá tato knihovna umožňuje tisknout znakové tabulky do konzole. Toho se využívá v případě možnosti *-r console*, kdy jsou výsledky benchmarku prezentovány do konzole. Knihovna *glob* je použita pro hledání souborů podle zadané masky. Toho se využívá při hledání souborů obsahujících kód pro testování. Poslední knihovnou je *json2csv*. Tato knihovna umožňuje jednoduše vytvářet CSV soubory při možnosti *-r console*. Zbylé závislosti definované v souboru *package.json* jsou testované knihovny.

Spouštěcím souborem je *init.js*, kde jsou definovány parametry spuštění a jejich validace. Při úspěšném spuštění je vytvořen objekt *Config*, ve kterém jsou předány parametry benchmarku. Poté jsou nalezeny veškeré testované knihovny a *js* soubory. Ty jsou umístěny ve složce *modules/metrics*. V této složce je také soubor *metric.js*, který obsahuje konstrukční funkci k objektu. Tento objekt definuje několik základních metod jako *run()*, *prepareDataForSerialize()*, *serializeImpl()*, *prepareDataForDeserialize()*, *deserializeImpl()* a *fullName()*.

Základní funkčnost měření času podle Kód 28 je implementována pomocí NodeJS funkce *process.hrtime()* v metodě *run*(). Velikost serializovaných dat je odečtena pomocí metody *length()*. Metoda *run()* dále postupně spouští jednotlivé zmíněné metody. Konkrétní knihovny pak vytvářejí objekty pomocí konstrukční funkce a v případě potřeby přetěžují některé zmíněné funkce.

Po skončení benchmarku jsou data obsažena v poli s objekty, které obsahují výsledné hodnoty. Podle parametru *-r* jsou data buď prezentována do konzole, nebo ve formě CSV souborů.

6.4 Docker

Aby bylo možné všechny benchmarky jednoduše spouštět bez nutnosti instalovat PHP, JAVU a NodeJS (JS) ve správné verzi a se správnou konfigurací, jsou všechny aplikace připraveny pro spuštění v technologii Docker. V podstatě se jedná o lehkou virtualizaci, kde jsou v předem připravených obrazech (Docker images) již nainstalována běhová prostředí pro nejrůznější programovací jazyky a další nezbytné programy a konfigurace. O běh těchto obrazů se stará tzv. Docker daemon. Při spuštění obrazu je z něj vytvořen kontejner (Docker container), jedná se v podstatě o instanci obrazu. Těchto kontejnerů může být z jednoho obrazu vytvořeno větší množství. Start kontejneru (~virtuálního stroje) je okamžitý, paměť RAM neukrajuje žádný další operační systém (kernel se sdílí) a ani nedochází k žádnému zpomalení při vlastním běhu programu (Augustýn, 2017). Tyto procesy se chovají stejně jako nativní procesy a lze je zobrazit spuštěním příkazu *ps* v prostředí Linuxu nebo pomocí správce úloh ve Windows atd. Docker podporuje celou řadu operačních systémů a jeho kontejnerové aplikace lze spustit takřka „kdekoliv“ (Vieux, 2017).

Obrazy lze nadále upravovat a konfigurovat. Základem těchto obrazů je nejčastěji operační systém Debian, popřípadě jiná Linuxová distribuce a konkrétní předinstalovaný software; existují i Windows obrazy. Seznam těchto obrazů lze najít v repositáři Docker HUB, jedná se například o obrazy: *node:7.7, maven:3.3-jdk-8, php:7.1-cli*. Právě na těchto obrazech jsou postaveny obrazy pro jednotlivé benchmarky. Konfigurace vlastních obrazů se provádí pomocí *Dockerfile* souborů.

Jedná se o textové soubory bez přípony, ve kterých je napsaný postup sestavování obrazu. Ukázka *Dockerfile* pro JavaScript benchmark (viz Kód 36):

Kód 36 - Dockerfile pro JavaScript

```
FROM node:7.7
COPY ./ /opt/benchmark/benchmark-nodejs
WORKDIR/opt/benchmark/benchmark-nodejs
RUN npm install
```

Na prvním řádku (viz Kód 36) je řečeno, z jakého obrazu budeme vycházet. V tomto případě se jedná o NodeJS ve verzi 7.7. Dále je třeba zajistit nahrání zdrojových kódů benchmarku do obrazu. Toho je docíleno pomocí příkazu *COPY*, který zkopíruje všechny soubory ze zadané cesty do cílového umístění. Na dalším řádku se přepneme do adresáře s benchmarkem pro JavaScript a poté spustíme příkaz *npm install,* který se postará o stažení všech závislostí na externích knihovnách.

Takto připravené *Dockerfile* soubory je potřeba sestavit (buildovat). Pro zjednodušení tohoto procesu je vytvořen script, který se o tuto činnost stará. Skript je napsán pro *shell* a testován v prostředí Linux. Název souboru nese označení *build-docker.sh*, lze jej spouštět s následující možností:

- *-l, --language <php|java|nodejs>* → sestavení obrazu pouze pro vybranou technologii.

Pokud je skript spuštěn bez parametrů, jsou sestaveny všechny tři obrazy. Jedinou závislostí tohoto skriptu je přítomnost samotného Docker.

Celková výhoda tohoto řešení tedy spočívá v jednoduchosti spouštění aplikací, není třeba instalovat vývojové platformy a správně je konfigurovat. Stačí pouze nainstalovat Docker a vše ostatní je jen záležitostí spouštění a sestavování obrazů. Běžící kontejnery jsou navíc spuštěny jako nativní procesy operačního systému. Jistou nevýhodou je nutnost stahování relativně velkého množství dat, které obrazy zabírají. Není výjimkou, že se jedná o stovky megabytů.

6.5 Hromadné spuštění testů

Použití Docker nám usnadní práci s nastavováním prostředí, ale spouštět jednotlivé benchmarky po jednom je stále neefektivní. Navíc pokud použijeme Docker, vyvstává otázka, s jakými parametry jednotlivé kontejnery spouštět, abychom se dobrali jednotlivých výsledků (CSV souborů). Z toho důvodu je vytvořen skript, opět pro shell, a tedy primárně pro Linux, který nám spouštění benchmarků usnadní. Jedná se o soubor s názvem *run-benchmarks.sh.*

Skript přebírá několik možností spuštění od jednotlivých benchmarků. Konkrétně se jedná o tyto:

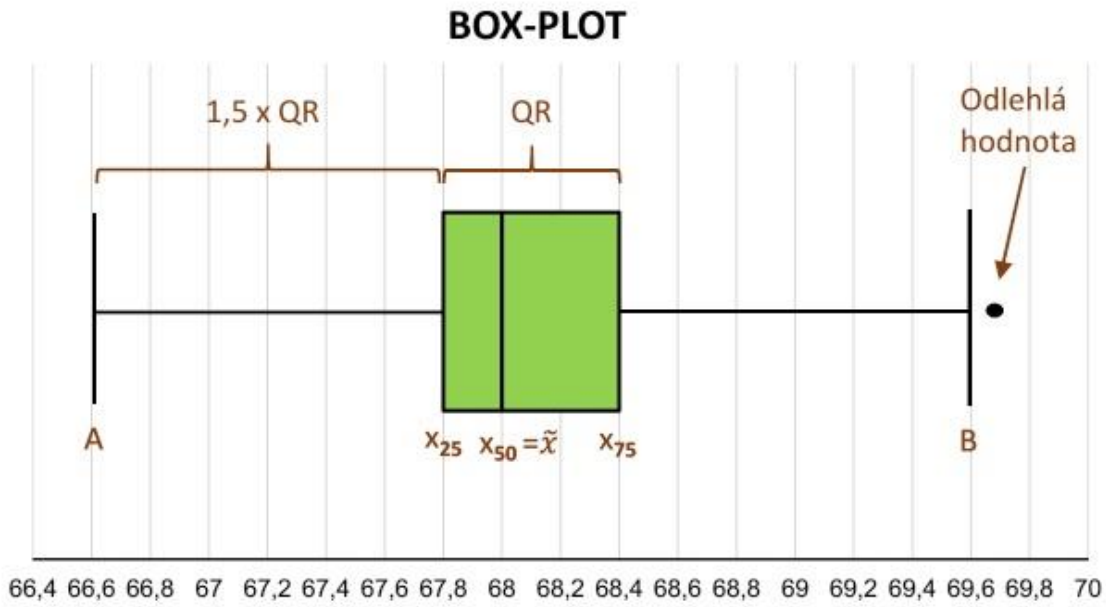
- *-o, --outer <n>* → vnější počet opakování (viz Kód 28).
- *-i, --inner <n>* → vnitřní počet opakování (viz Kód 28).
- *-f, --format <native|json|xml|protobuf|avro|msgpack>* → testuj pouze zadaný formát.
- *-l, --language <php|java|nodejs>* → testuje pouze zvolený jazyk.
- *-c, --chatty* → při testování vypisuje aktuální činnost.

Po proběhnutí benchmarků skript dále vytváří několik extra souborů, konkrétně soubor *combined-summarize.csv*, který obsahuje data ze všech **-summarize.csv* souborů. Dále skript vytváří dva soubory, které obsahují sjednocená data pro serializaci a deserializaci, konkrétně soubory *combined-serialize.csv* a *combined-deserialize.csv*. Navíc jsou tyto dva soubory rozděleny každý do tří souborů, přičemž každý z nich obsahuje třetinu dat, data jsou seřazena podle průměrného času od nejmenšího po největší.

Pokud je na hostitelském počítači nainstalován program Gnuplot, jsou navíc z těchto souborů vygenerovány grafy ve formě PNG obrázků. Gnuplot je nástroj, který pomocí série příkazů dokáže generovat nejrůznější množství grafů z textových dat. Konkrétně ze souboru *combined-summarize.csv* jsou vygenerovány tři sloupcové grafy seřazené od nejmenších hodnot po největší, jeden pro serializaci, jeden pro deserializaci a jeden pro velikost serializovaných dat. Pro lepší přehlednost je na ose X zvoleno logaritmické měřítko. Ze souborů *combined-serialize.csv* a *combined-deserialize.csv* jsou vygenerovány krabicové (box) grafy. S cílem poskytnout lepší

přehlednost jsou ještě vytvářeny tři další soubory (obrázky), kde každý soubor obsahuje třetinu dat (krabicových grafů) ze všech testovaných.

Krabicový graf je jednou z grafických metod používaných ve statistice, která umožňuje posouzení dat pomocí kvartilů (Dudek, 2017). Ukázka krabicového grafu (viz Graf 2):



Graf 2 - Popis krabicového grafu

(zdroj: http://kvalita-jednoduse.cz/wp-content/uploads/2017/02/Box1.jpg)

Krabicový graf tedy zobrazuje několik zajímavých hodnot v jednom obrázku, konkrétně odlehlé hodnoty, první kvartil, medián, druhý kvartil, kvartilové rozpětí, minimum a maximum bez odlehlých hodnot.

Pro spuštění benchmarků v prostředí Linux stačí pouze spustit dva skripty. Jeden slouží pro sestavení Docker obrazů a druhý pro hromadné spuštění a vytvoření grafů.

7 Vyhodnocení

V následujících kapitolách jsou prezentovány výsledky benchmarků s různým nastavením spuštění. Výsledky jsou dále porovnávány s některými jinými testy od dalších autorů. Pokud není řečeno jinak, probíhá veškeré testování na počítači s touto hardwarovou a softwarovou výbavou:

- Procesor: Intel Core I7-2600k 3.4 GHz.
- Paměť: 8 GB DDR3.
- Operační systém: Debian 8
- PHP: 7.1
- JDK: 8
- NodeJS: 7.7

Celkem je v testu zahrnuto okolo 40 knihoven, přičemž ne všechny podporují zpětnou deserializaci, zejména pak některé XML knihovny. Kritéria pro zahrnutí knihovny do benchmarku jsou: počet stažení (podle npm, Maven, Composer), počet hvězdiček na githubu, počet založených problémů u repositáře, aktuálnost kódu. Ne vždy jsou splněna všechna kritéria. U knihoven je vždy použita nejvyšší stabilní verze, která byla dostupná v době testování. Některé knihovny nemají označenou stabilní verzi, proto jsou v testu i knihovny ve verzi beta, popřípadě RC[12].

Některé knihovny jsou v testu zahrnuty vícekrát, jelikož poskytují více přístupů k serializaci. Nejčastěji knihovny k zmapování dat a názvu proměnných využívají reflexi, některé ale umožňují definovat vlastní mapování. Takové knihovny jsou v benchmarku označeny slovem „Custom“.

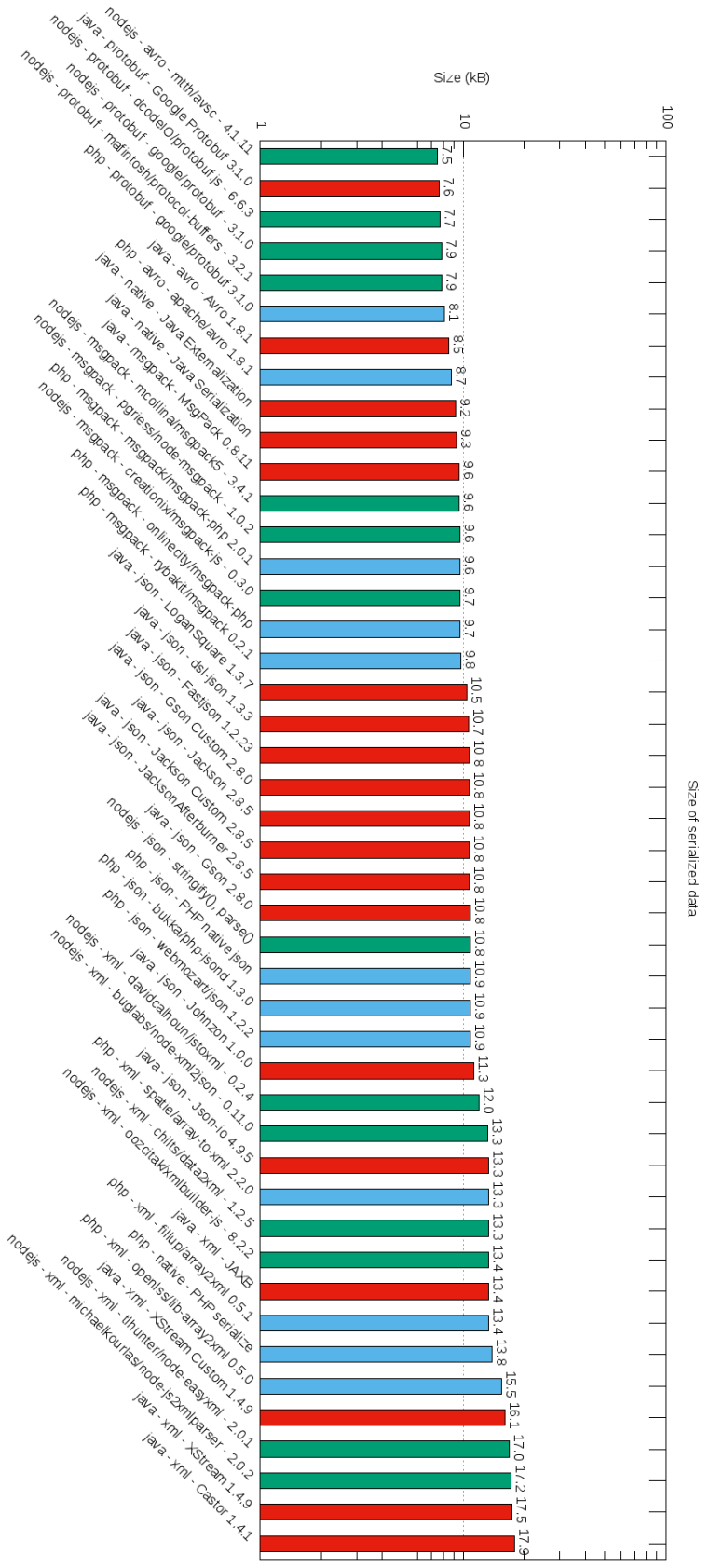
7.1 Datová náročnost serializace

Prvním z testovaných vlastností serializačních formátů je velikost výsledných dat. Teoreticky by měly formáty, které vyžadují schéma, dopadnout v tomto testu

[12] RC – release candidate, kandidát na vydání stabilní verze. Obvykle následuje po vydání alfa a beta verze.

nejlépe, jelikož nepotřebují ukládat všechna data. Část dat je definována ve schématu, které je nedílnou součástí serializačního procesu. Jedná se především o formáty Protocol Buffers a Avro. Nejhůře by na tom měl být formát XML, který patří k nejupovídanějším (Maeda, 2011).

Výsledná velikost dat je vždy naměřena jen na jednom průchodu serializace. Počet vnějších a vnitřních iterací (Kód 28) je z hlediska tohoto testu irelevantní. Podoba testovacích dat je zmíněna v kapitole Testovací data.



Graf 3 - Velikost serializovaných dat

(zdroj: autor)

Graf 3 zobrazuje velikost serializovaných dat pro všechny testované knihovny. Čím menší hodnota, tím lépe. Výsledné hodnoty jsou zaokrouhleny na desetiny v jednotkách kilobytů (kB). Zeleně jsou obarveny knihovny v NodeJS (JavaScript), červeně JAVA a modře PHP.

Jak bylo předesíláno, na prvních místech se umístily právě formáty Avro a Protobuf a nejhůře dopadl formát XML. Konkrétně první místo obsadil formát Avro pro NodeJS v implementaci *mtth/avsc* ve verzi 4.1.11, který dosáhl hodnoty 7.5 kB, což je o 10 kB méně než nejhorší knihovna v tomto testu JAVA XML Castor 1.4.1. Z výsledků lze vypozorovat, že obecně si nejlépe vedou formáty Avro a Protobuf, poté následuje MsgPack, za ním JSON a na posledním místě XML. To potvrzuje například benchmark „Benchmarks“ (Monsch, 2016).

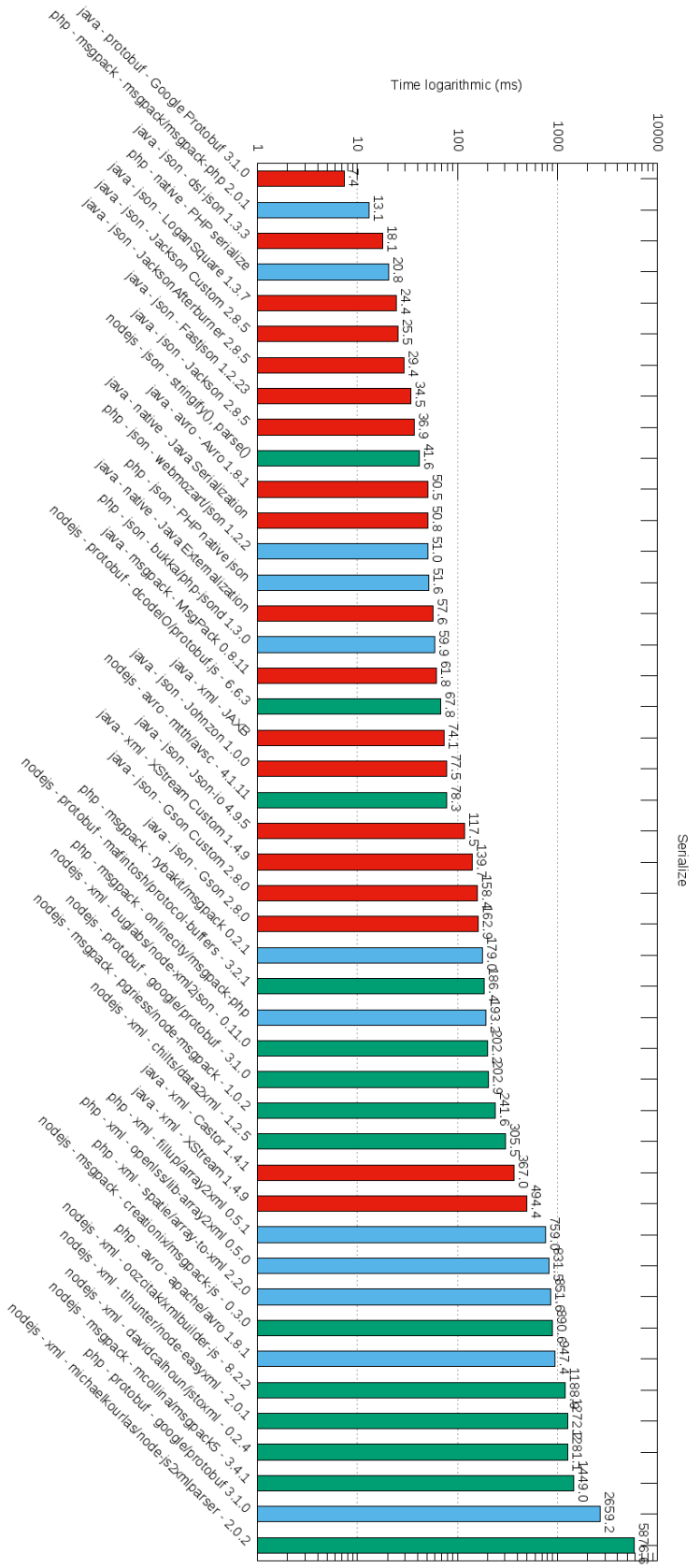
Zajímavým poznatkem je, že ačkoliv některé knihovny implementují stejný formát, dosahují rozdílných výsledků. To lze například vysvětlit tím, že neexistuje jednoznačný převod dat mezi kódem programu a jeho serializací. Vezmeme-li například formát XML, máme hned několik možností, jak zakódovat pole hodnot, především pak volba názvů XML elementů. Využít můžeme název atributu, který drží referenci na pole jako název jednotlivých položek pole, a k nim přidáme další znaky, tak aby nedocházelo ke kolizi názvů, například: číslo, pomlčku a číslo aj. Položky můžeme navíc obalit jedním kořenovým elementem. Dále máme možnost použít atributy. Všechny tyto možnosti mění výslednou velikost dat. Obdobně tento problém můžeme aplikovat i na zbylé formáty. Navíc u formátu XML a JSON, knihovny obvykle umožnují přepínání formátování výstupních dat mezi tzv. „pretty“ a úsporným režimem. Režim „pretty“ přidává do výstupu bílé (whitespace) znaky jako mezery, tabulátory a entery, a to pro lepší čitelnost dat. Úsporný režim naopak tyto znaky odstraňuje. Pro všechny testované knihovny je použit úsporný režim, pokud je toto nastavení dostupné.

7.2 Časová náročnost serializace

Dalším testovaným kritériem je čas potřebný k serializování dat. Jak již bylo zmíněno, samotné časové údaje sami o sobě nejsou nejdůležitější, jelikož se budou lišit v závislosti na hardwaru počítače, v němž je benchmark spuštěn. Podstatné je

vzájemné porovnání naměřených údajů napříč formáty, knihovnami a programovacími jazyky.

Benchmark je proveden nad stejnými testovacími daty jako předešlý test, viz Testovací data. Tentokrát je ale počet iterací značně zvýšen. Konkrétně jsou stejná data tisíckrát serializována pro jednu knihovnu a celý tento proces je navíc stokrát zopakován. Tedy vnitřní počet opakování je 1000 a vnější je roven 100, viz Kód 28. Graf 4 zobrazuje sloupcový graf výsledků měření.



Graf 4 - Sloupcový graf serializace všech knihoven

(zdroj: autor)

Čím menší výsledná hodnota, tím lépe. Časy jsou zaokrouhleny na jedno desetinné místo v jednotkách milisekundy (ms). Jelikož časy mezi nejlepšími a nejhoršími výsledky jsou propastné, je na ose Y zvoleno logaritmické měřítko pro lepší přehlednost. Barvy opět korespondují s technologiemi: červeně JAVA, modře PHP a zeleně NodeJS (JavaScript).

Na prvním místě se umístil formát Protobuf implementovaný v JAVĚ s časem 7,4 ms, jedná se o oficiální implementaci. Zajímavé ovšem je, že stejný formát implementovaný v PHP je na předposledním místě s časem 2,6 s, opět se jedná o oficiální implementaci. Testovaná verze 3.1.0 je první stabilní verzí pro PHP, je tedy možné, že v této verzi se hledělo především na samotnou funkčnost, a nikoliv na výkon. Tato konkrétní knihovna navíc nebyla v době testování dostupná jako nativní rozšíření PHP, ale pouze jako závislost Composeru, což může mít také dopad na výkon. Při porovnání implementace v JS je vidět, že ani tady oficiální implementace s časem 202,9 ms není nejrychlejší. Poráží ji *dcodeIO/Protobuf.js* s časem 67,8 ms. I v tomto případě to může být způsobeno tím, že JS není oficiálně podporován příliš dlouho, první stabilní verze vyšla na konci léta 2016.

Velice dobrého výsledku také dosáhl formát MsgPack knihovna pro PHP *msgpack/msgpack-php* s časem 13,1 ms. Jedná se o oficiální nativní rozšíření PHP oproti dvěma dalším knihovnám v PHP, které jsou naprogramovány v jazyce PHP. Ani implementace v JAVĚ si nevede špatně s časem 62 ms. Knihovny pro JS jsou už o několik řádů výš, konkrétně knihovna *pgriess/node-msgpack* s časem 242 ms.

Celkově se v testu dobře umístila JAVA, a to zejména JSON knihovny, které překonaly i nativní JavaScriptovou funkci *JSON.stringify()*. Výsledky JSON knihoven v JAVĚ lze porovnat s benchmarkem „Performance testing of serialization and deserialization of Java JSON libraries“ (Renaud, 2017), kde na prvních třech pozicích jsou taktéž JSON knihovny: *dsl-json* na prvním místě, *LoganSquare* na druhém a varianty použití knihovny *jackson* na třetím.

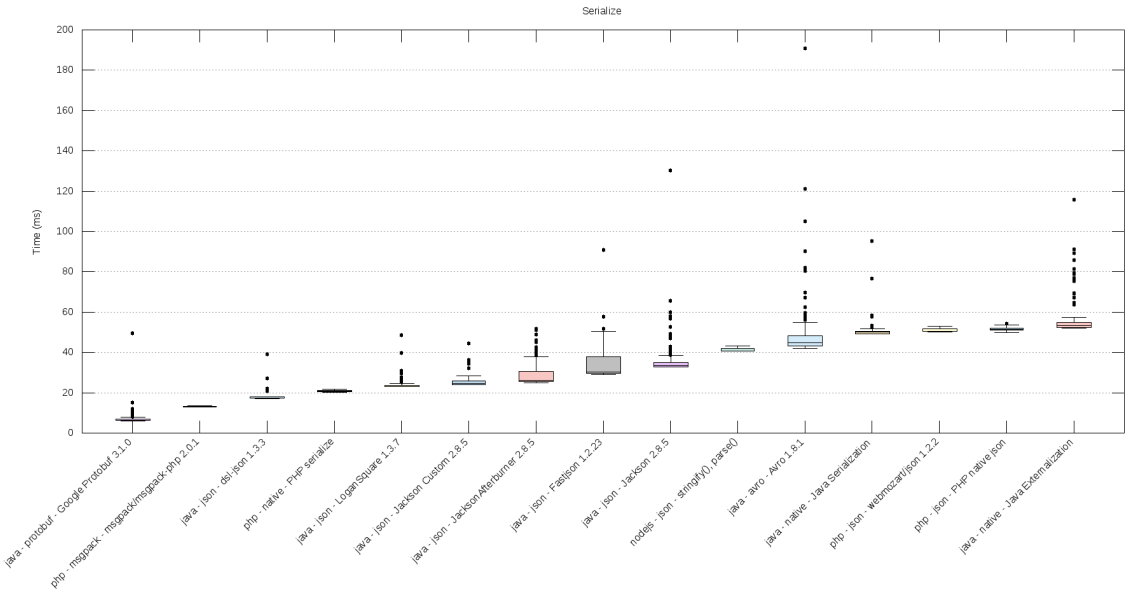
Výsledky také potvrdili, že JSON je obecně rychlejší než XML. V benchmarku žádná XML knihovna nepředstihla ani jednu JSON knihovnu. Tento výsledek potvrzují i benchmarky „Comparison of JSON and XML Data Interchange Formats: A Case

Study“ (Nurseitov, a další, 2009) a „JSON vs. XML: Some hard numbers about verbosity“ (Pragmateek, 2013).

Ne všechny benchmarky korespondují s výsledky tohoto benchmarku. Například benchmark pro JAVU „Benchmark comparing serialization libraries on the JVM“ (Saloranta, 2017) nestaví formát Protobuf na první místo, ale vždy až za knihovnu *dsl-json* a formát MsgPack.

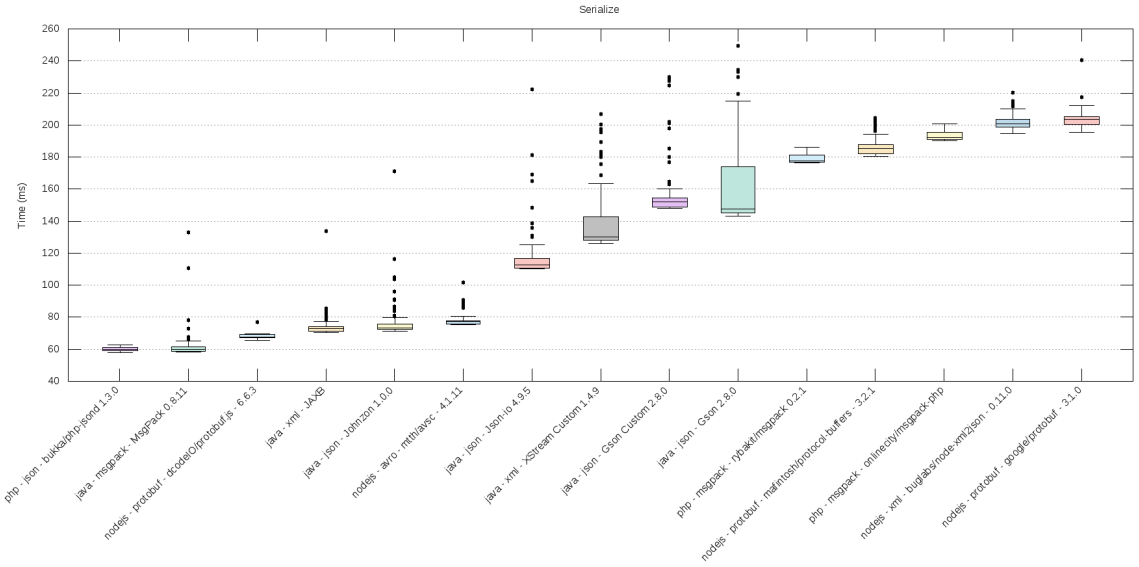
Porovnání s ostatními výše zmíněnými benchmarky je obtížné, jelikož u nich často chybí označení verze použitých knihoven, a navíc žádný z benchmarků není multiplatformní (testuje více programovacích jazyků).

Dále je z výsledků měření vytvořen krabicový graf. Jedná se právě o naměřených sto hodnot ve vnějších iteracích. Graf je pro větší přehlednost dále rozdělen na tři podgrafy: Graf 5, Graf 6, Graf 7.



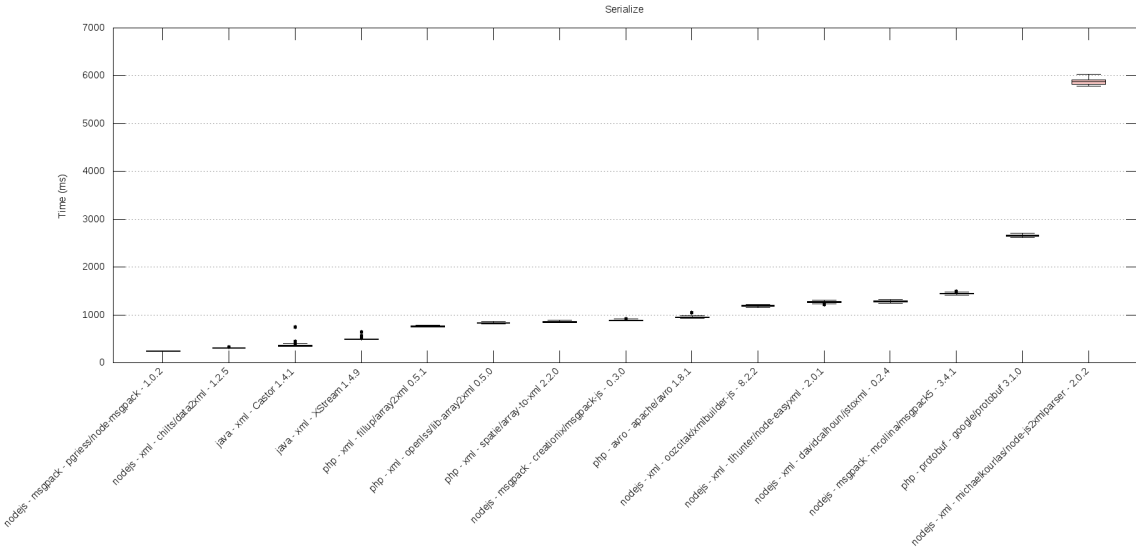
Graf 5 - Krabicový graf serialize všech knihoven 1/3

(zdroj: autor)



Graf 6 - Krabicový graf serialize všech knihoven 2/3

(zdroj: autor)



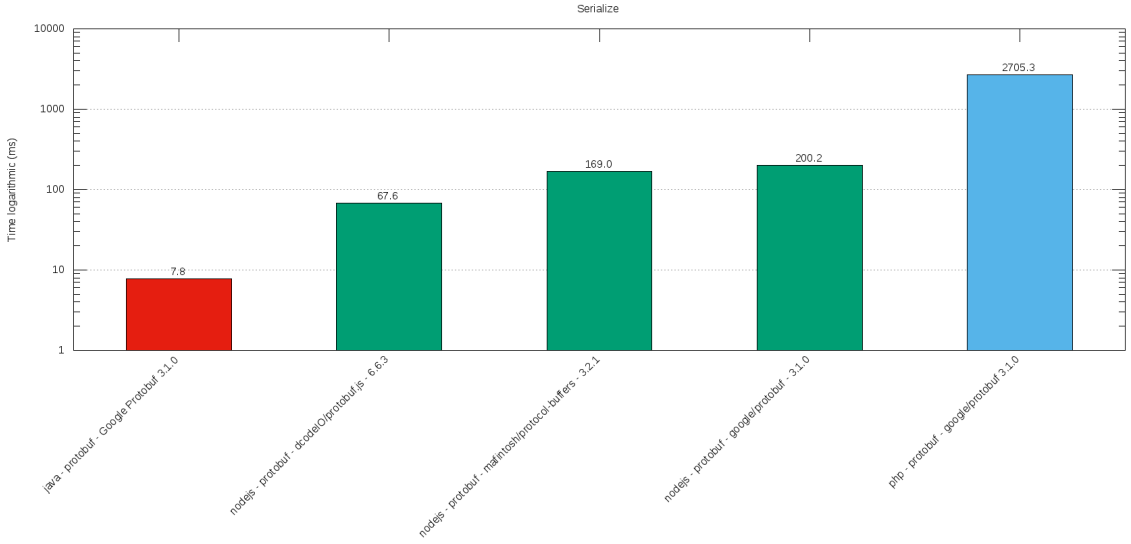
Graf 7 - Krabicový graf serialize všech knihoven 3/3

(zdroj: autor)

Jednotlivé knihovny jsou seřazeny podle výsledků (viz Graf 4) od nejlepšího k nejhoršímu a poté rozděleny do tří skupin: Graf 5, Graf 6, Graf 7. Jednotky na ose Y jsou stále v milisekundách (ms), ale měřítko již není logaritmické.

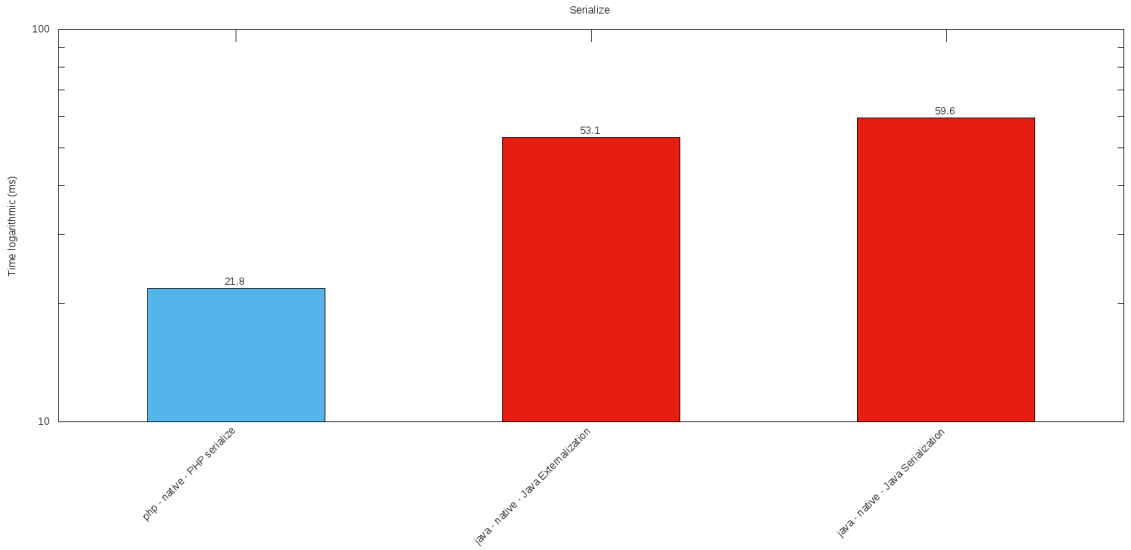
Grafy potvrzují, že největší rozptyl hodnot má JAVA, u níž dojde k ustálení hodnot (warm-up) až po několika iteracích, viz kapitolu Implementace v Java. U knihoven napsaných v PHP nebo JS toto chování až na výjimky nepozorujeme.

Dále následuje ukázka grafů pro jednotlivé formáty: Protobuf Graf 8, nativní formáty Graf 9, MsgPack Graf 10, Json Graf 11, Avro Graf 12 a Xml Graf 13. Pro každý formát byl spuštěn nový test se stejným počtem iterací, proto je možné, že se některé hodnoty budou lišit oproti výsledkům viz Graf 4.



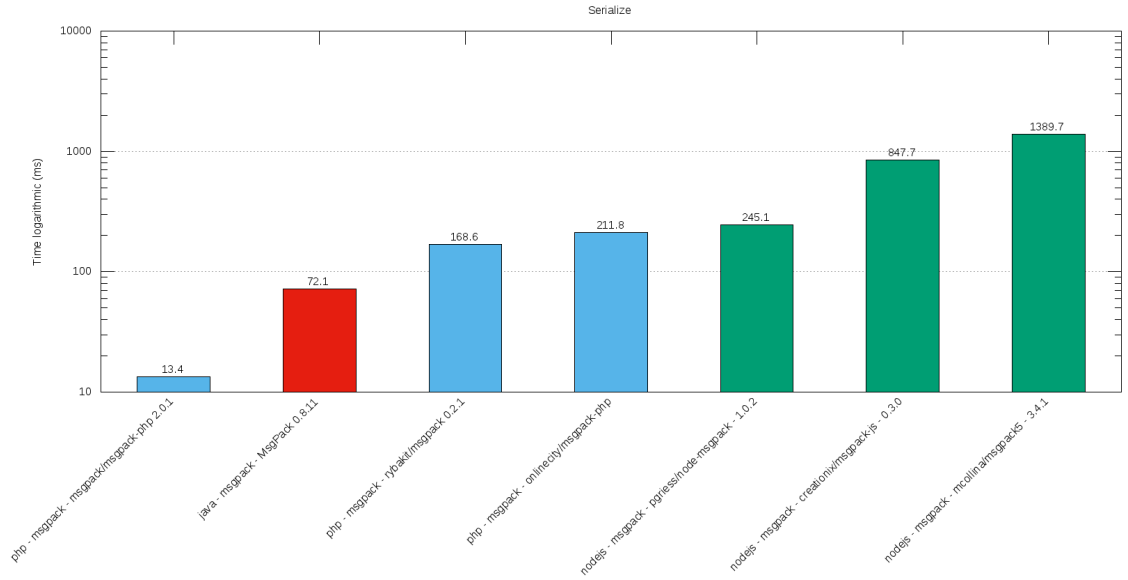
Graf 8 - Sloupcový graf serializace formátu Protobuf

(zdroj: autor)



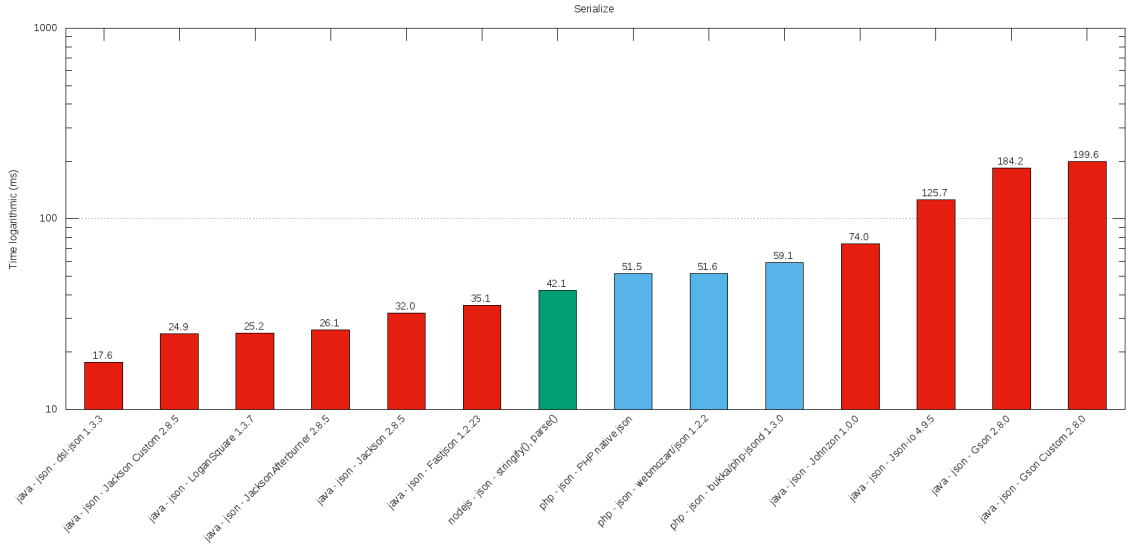
Graf 9 - Sloupcový graf serializace nativních formátů

(zdroj: autor)



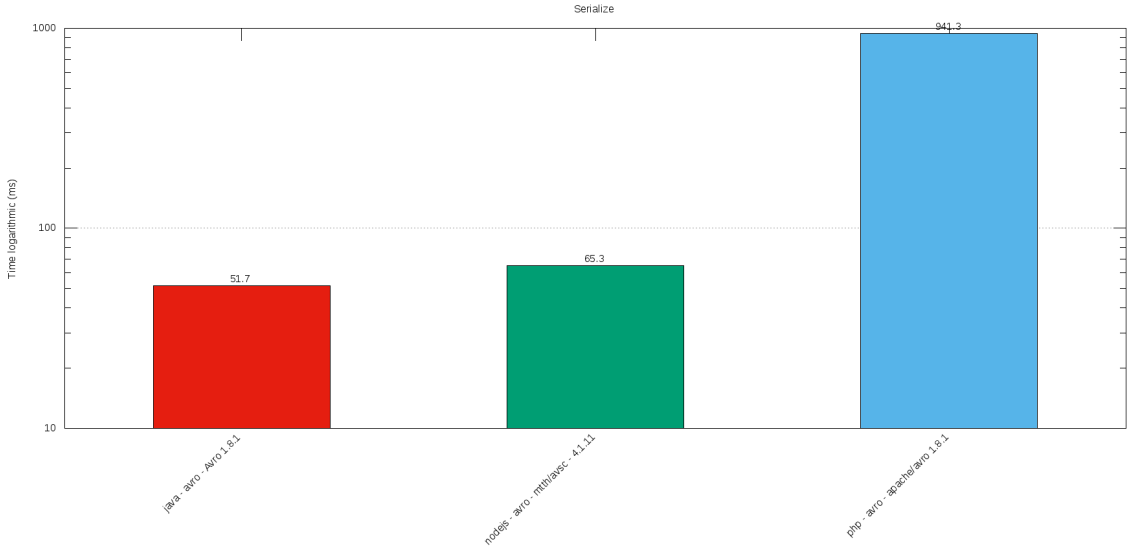
Graf 10 - Sloupcový graf serializace formátu MsgPack

(zdroj: autor)



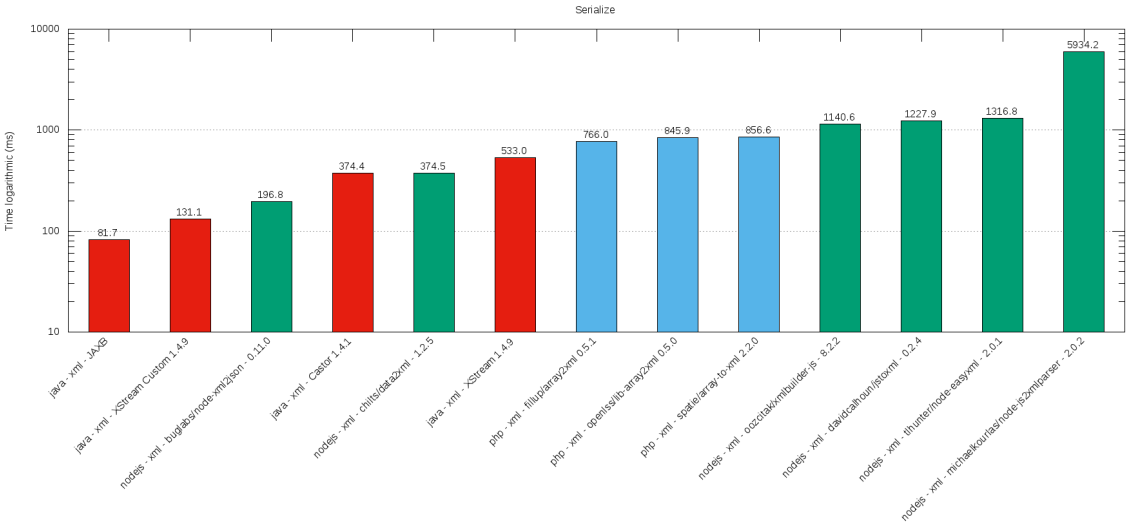
Graf 11 - Sloupcový graf serializace formátu Json

(zdroj: autor)



Graf 12 - Sloupcový graf serializace formátu Avro

(zdroj: autor)



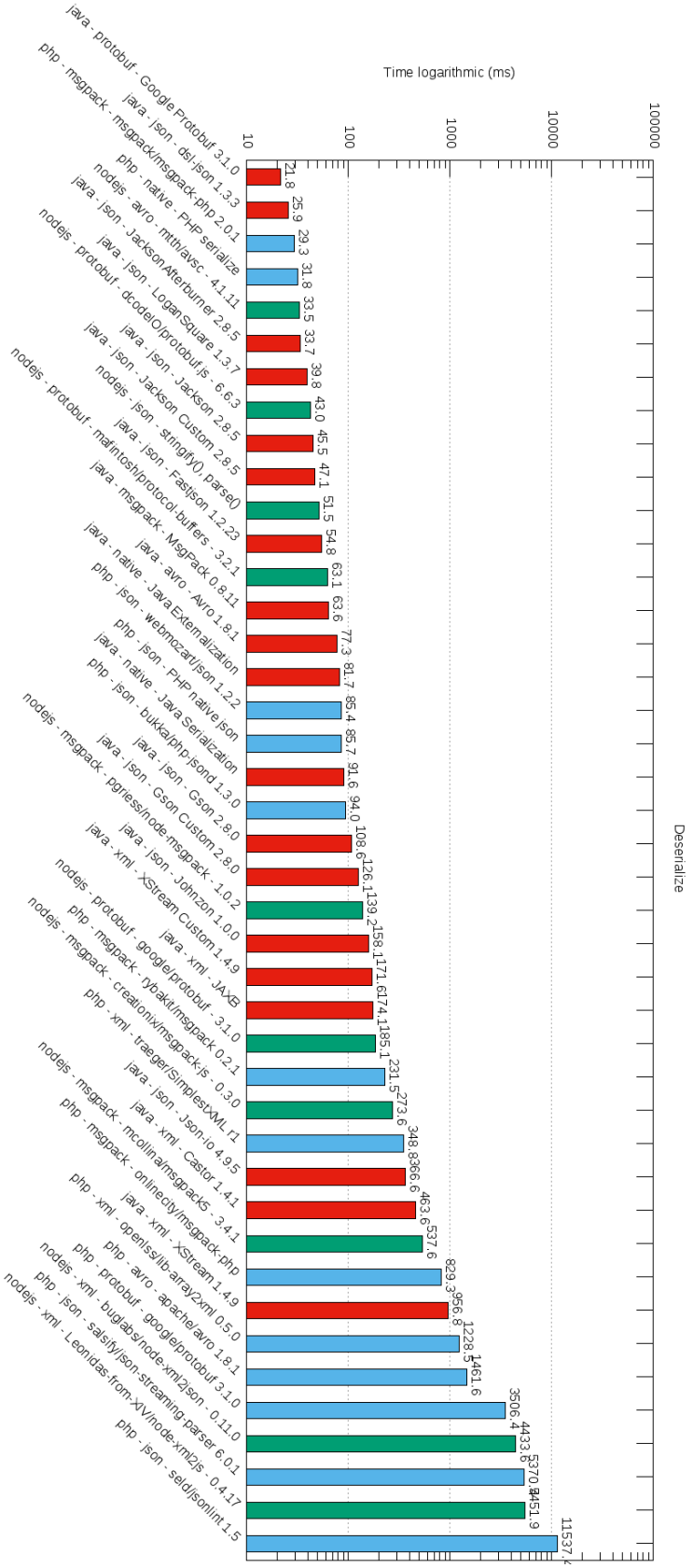
Graf 13 - Sloupcový graf serializace formátu Xml

(zdroj: autor)

Veškeré uvedené grafy jsou generovány automaticky pomocí skriptu hromadného spouštění benchmarku.

7.3 Časová náročnost deserializace

Posledním testovaným údajem je čas potřebný k deserializaci dat. Stejně jako v předešlém testu jsou použita stejná testovací data se stejným počtem opakování. Výsledek benchmarku (viz Graf 14).



Graf 14 - Sloupcový graf deserializace všech knihoven

(zdroj: autor)

Časy jsou opět zaokrouhleny na desetiny v milisekundách (ms) s logaritmickým měřítkem. Čím menší dosažený čas, tím lépe. Barvy korespondují s technologiemi: červeně JAVA, modře PHP a zeleně NodeJS (JavaScript).

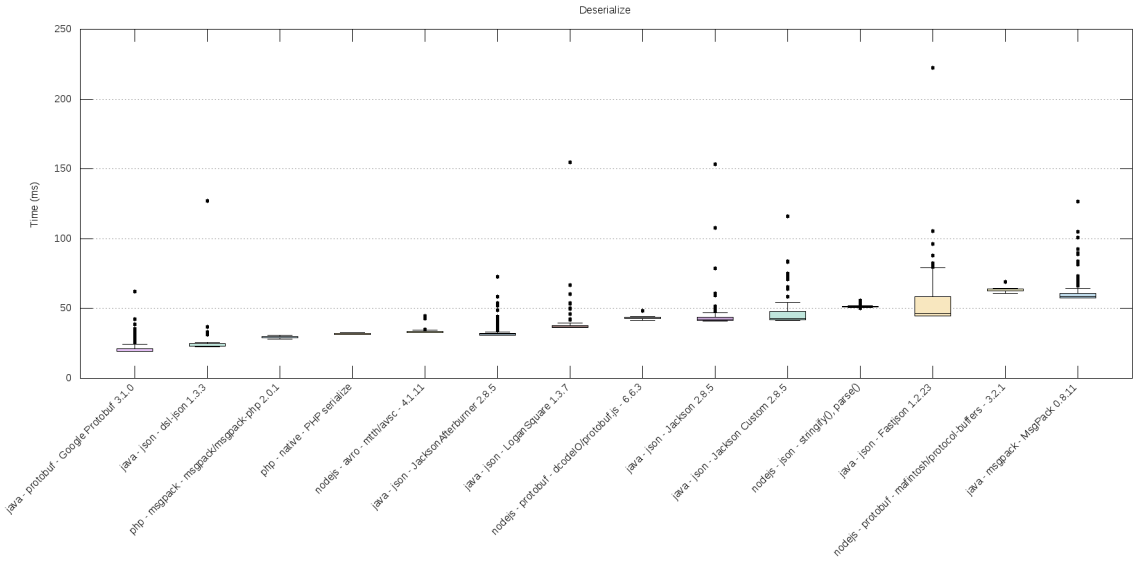
Na prvním místě se opět umístil formát protobuf implementovaný v JAVĚ ve verzi 3.1.0 s časem 21,8 ms. Na druhé místo se dostala knihovna *dsl-json 1.3.3* s časem 25,9 ms, která přeskočila *msgpack-php 2.0.1*, umístěný na druhé pozici v benchmarku serializace. Celkově si opět dobře vedli JSON knihovny implementované v JAVĚ. Opět je možné provést porovnání s „Performance testing of serialization and deserialization of Java JSON libraries" (Renaud, 2017), kde první místa patří knihovnám *dsl-json, Jackson a LoganSquare.*

Stejně jako u testu serializace i u deserializace ne všechny benchmarky korespondují s naměřenými výsledky. I v tomto případě benchmark „Benchmark comparing serialization libraries on the JVM" (Saloranta, 2017) nestaví Protobuf na první místo, ale až za zmíněnou *dsl-json* knihovnu nebo MsgPack formát.

Obecně lze z výsledků vypozorovat, že čas potřebný k deserializaci dat je o něco větší než čas potřebný k serializaci dat. Tuto skutečnost potvrzují i ostatní benchmarky jako například „Benchmarking BSON, JSON, and Native Serializing in PHP" (Suarez, 2016) nebo „PHP Serialization Benchmarks" (Sági-Kazár, 2017).

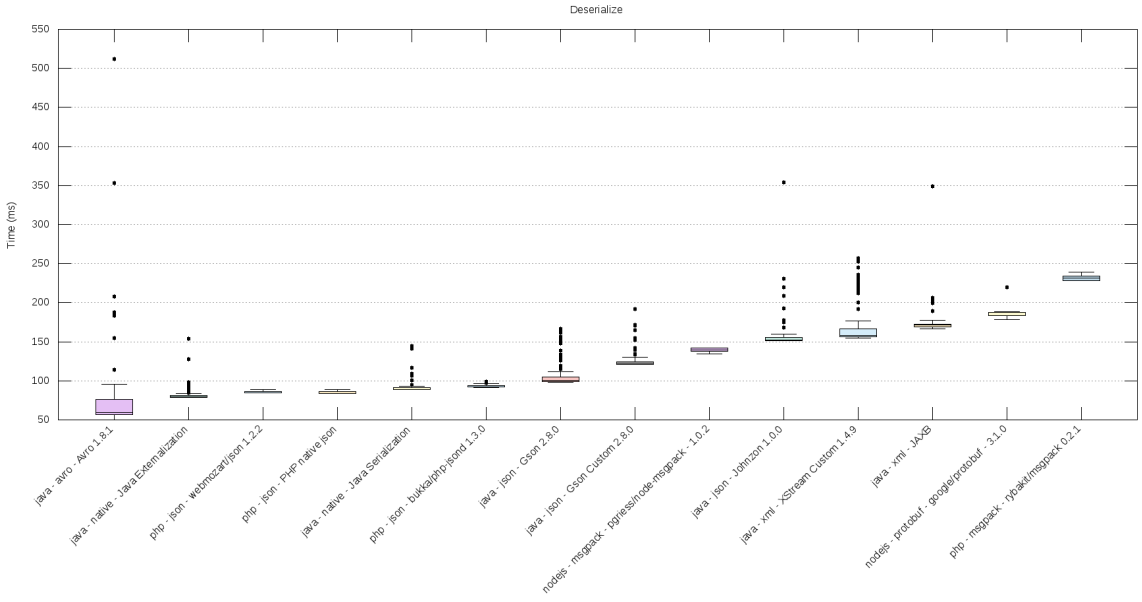
Jak již bylo řečeno, porovnání s ostatními zmíněnými benchmarky je obtížné, jelikož často chybí popis verzí knihoven nebo běhového prostředí, benchmarky často také testují jiné formáty nebo knihovny.

Dále je z výsledků měření vytvořen krabicový graf, který je rozdělen na tři podgrafy: Graf 15, Graf 16, Graf 17.



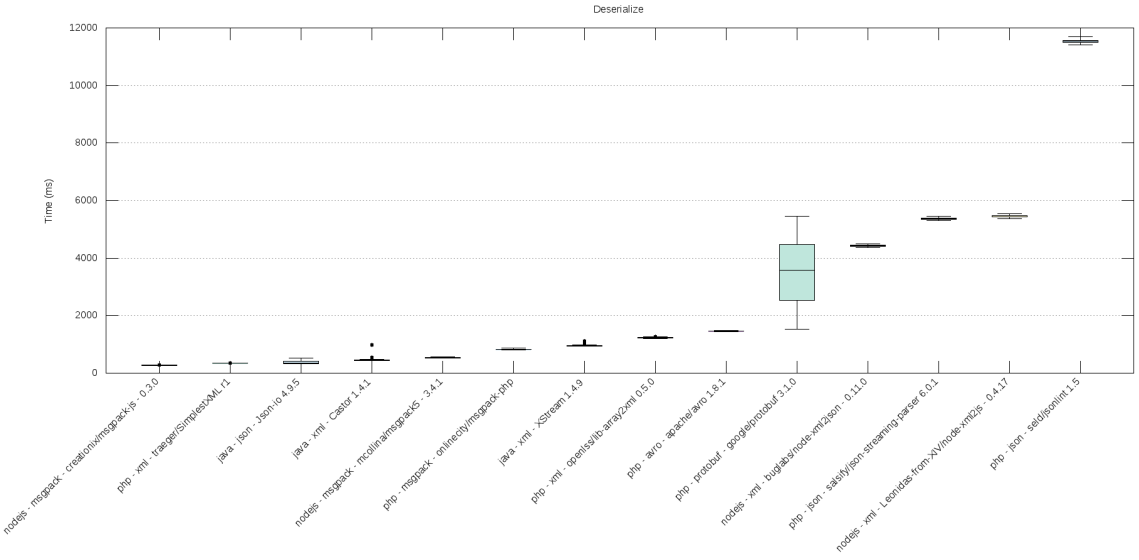
Graf 15 - Krabicový graf deserialize všech knihoven 1/3

(zdroj: autor)



Graf 16 - Krabicový graf deserialize všech knihoven 2/3

(zdroj: autor)

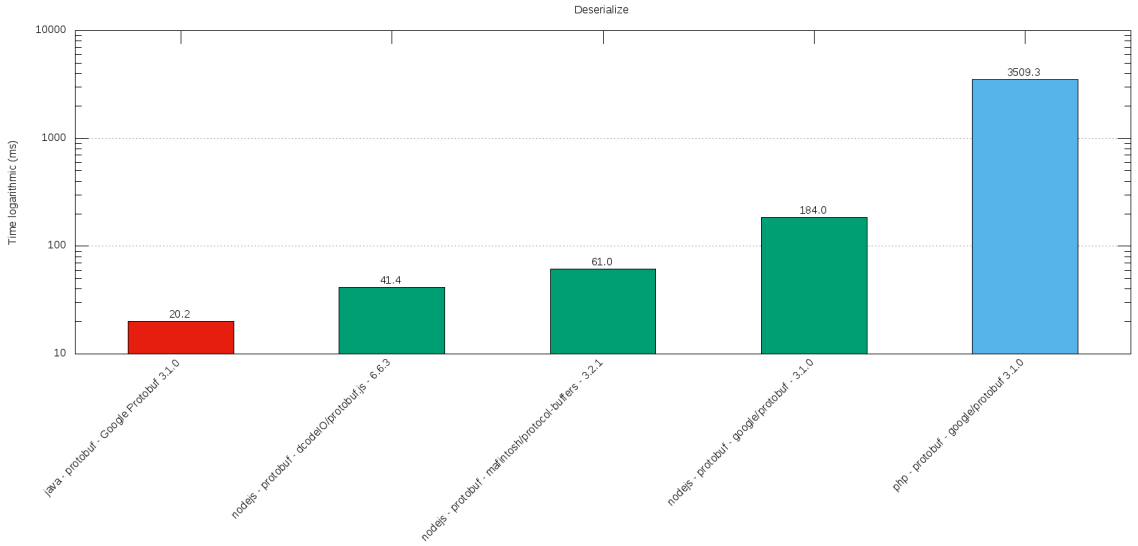


Graf 17 - Krabicový graf deserialize všech knihoven 3/3

(zdroj: autor)

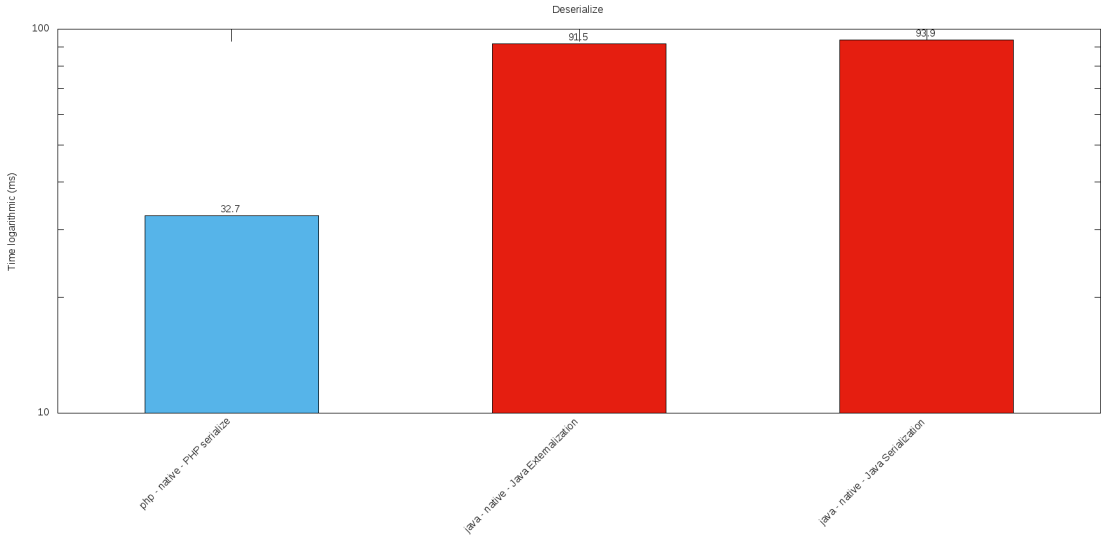
Z krabicových grafů je opět patrné, že největší rozptyl hodnot má JAVA. Jedinou výjimkou je implementace protobuf formátu v PHP. Tato knihovna byla jedna z nejpomalejších v testu serializace a ani v testu deserializace nedosahuje nejlepších výsledků. Zejména v kontrastu toho, že stejná oficiální implementace pro JAVU se umístila na prvním místě. Jak bylo řečeno v předchozí kapitole, jedná se o první stabilní verzi knihovny a důraz byl zřejmě kladen především na funkčnost.

Pro lepší srovnání jednotlivých formátů byl ještě spuštěn benchmark pro jednotlivé formáty napříč technologiemi. Viz Graf 18 Protobuf, Graf 19 nativní formáty, Graf 20 MsgPack, Graf 21 JSON, Graf 22 Avro, Graf 23 XML. Pro každý formát byl benchmark spuštěn znova, proto je možné, že se hodnoty výsledků budou lišit (viz Graf 14).



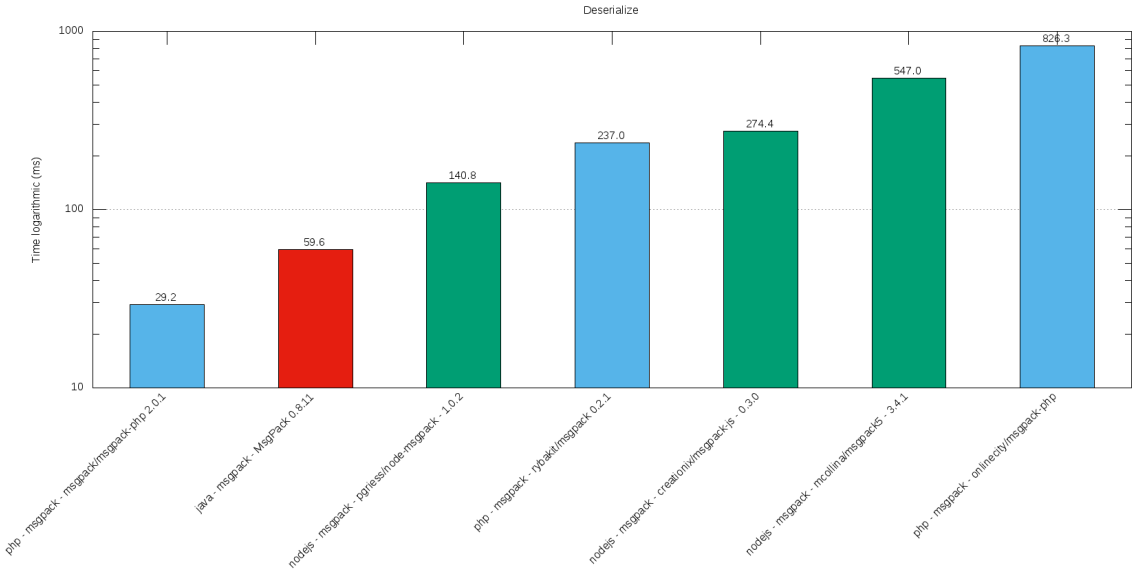
Graf 18 - Sloupcový graf serializace formátu Protobuf

(zdroj: autor)



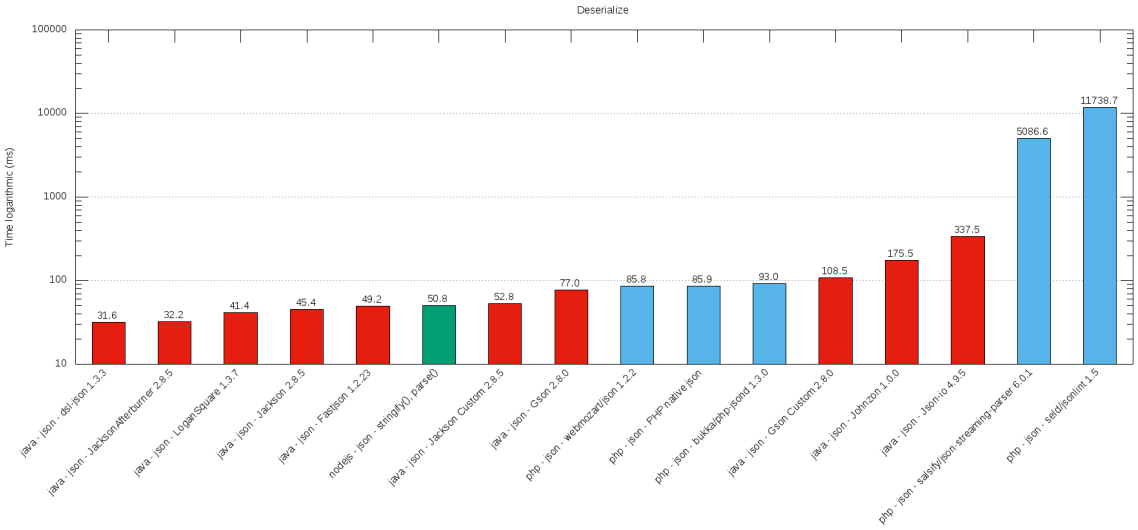
Graf 19 - Sloupcový graf serializace nativních formátů

(zdroj: autor)



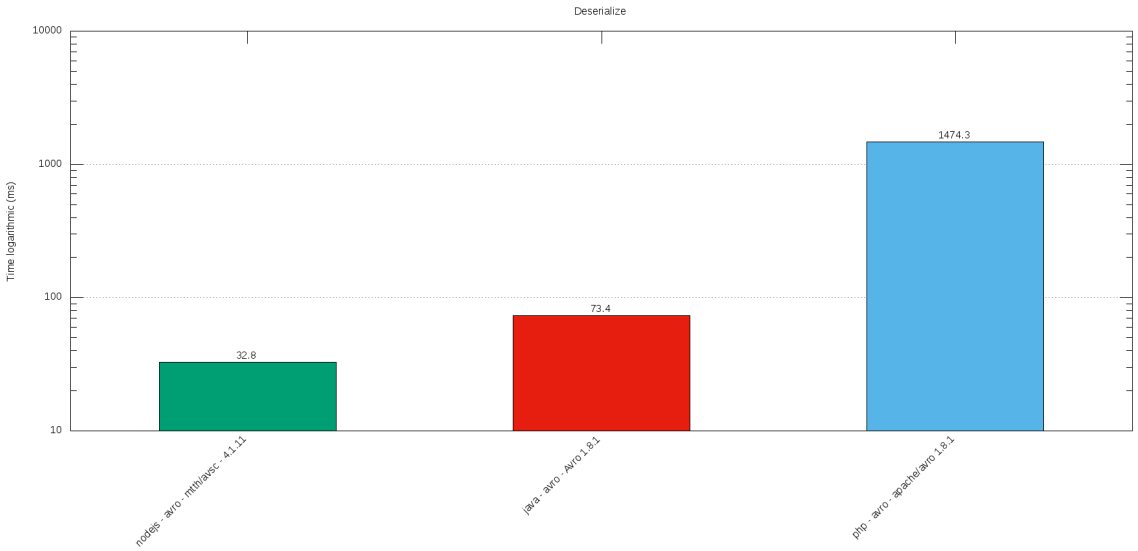
Graf 20 - Sloupcový graf serializace formátu MsgPack

(zdroj: autor)



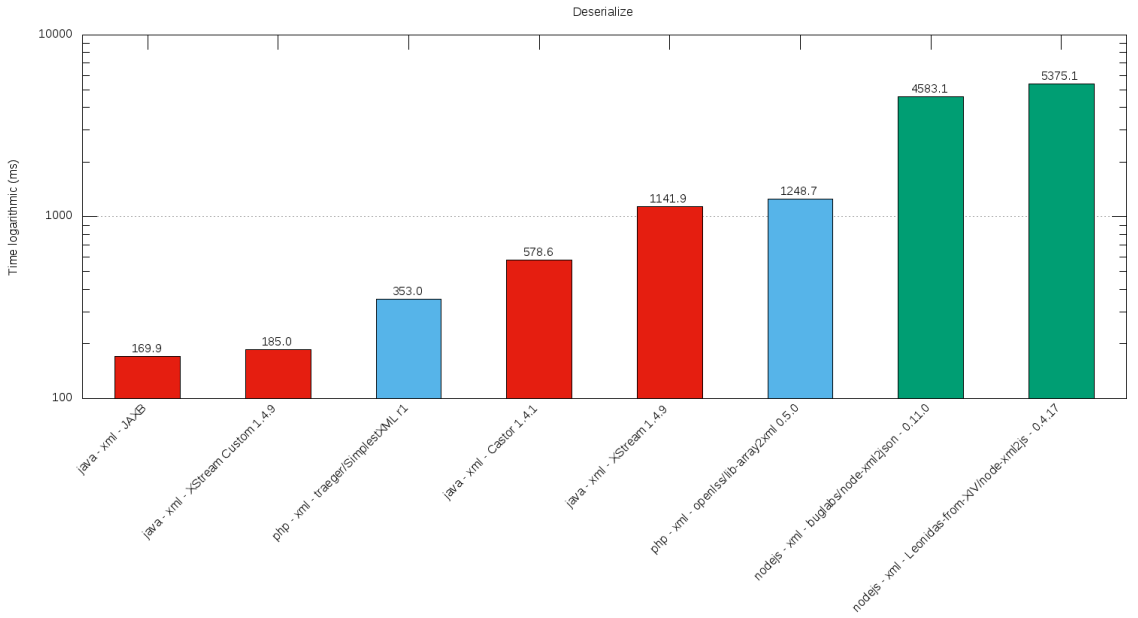
Graf 21 - Sloupcový graf serializace formátu JSON

(zdroj: autor)



Graf 22 - Sloupcový graf serializace formátu Avro

(zdroj: autor)



Graf 23 - Sloupcový graf serializace formátu Xml

(zdroj: autor)

8 Závěry a doporučení

Cílem práce bylo otestovat různé datové formáty pro výměnu dat, které se využívají typicky u webových služeb. Porovnat jejich časovou a paměťovou náročnost napříč různými programovacími jazyky.

Nejprve jsou v práci vysvětleny základní pojmy jako serializace a deserializace dat. Jejich základní varianty, užití, výhody a nevýhody. Poté je v práci popsáno pět formátů pro serializaci dat (Avro, JSON, MsgPack, Protobuf, XML) a také nativní formáty programovacích jazyků. Formáty jsou stručně představeny, jsou popsány jejich výhody a nevýhody. Dále následuje stručný popis programovacích jazyků, ve kterých proběhlo testování, konkrétně se jedná o JAVU, PHP a JavaScript.

Hlavní částí práce je návrh a implementace aplikace, která umožní jednoduché spuštění benchmarku, jehož výstupem je porovnání jednotlivých knihoven z hlediska časové a paměťové náročnosti. Testování proběhlo ve třech zmíněních jazycích pro pět na platformě nezávislých formátů. Do testu jsou též zahrnuty nativní serializace jazyků JAVY a PHP. Celkem bylo otestováno okolo čtyřiceti knihoven.

Z hlediska paměťové náročnosti nejlepších výsledků obecně dosahují formáty Avro a Protobuf. Jedná se o binární formáty, které vyžadují definici schématu. Část informací není tedy potřeba serializovat, jelikož jsou součástí schématu, z čehož vyplývá menší velikost výsledných dat. Nejhorších výsledků pak obecně dosáhly XML knihovny.

Časová náročnost je rozdělena na čas potřebný k serializaci a deserializaci dat. Nicméně v obou případech v benchmarku vyhrála knihova *Google/Protobuf* ve verzi *3.1.0* implementována v JAVĚ, za ní následují knihovny *msgpack/msgpack-php 2.0.1* v PHP nebo *dsl-json 1.3.3* opět v JAVĚ. Nicméně stejná knihovna (jako zmíněná vítězná knihovna *Google/Protobuf*) implementovaná v PHP se umístila na posledních pozicích. Velké rozdíly jsou i v rámci knihoven napsaných pro jeden jazyk, které implementují jeden formát. Příkladem může být formát MsgPack v PHP, v němž je jedna knihovna až stokrát rychlejší než zbylé dvě. Obdobné výsledky lze pozorovat i u jiných knihoven a formátů. Nelze tedy jednoznačně určit vítěze

formátu napříč jazyky. Obecně lze ale říci, že v průměru mezi jazyky si dobře vede formát JSON, naopak formát XML patří k nejpomalejším. U ostatních formátů se výsledky velice liší v závislosti na jazyku a konkrétní knihovně.

V budoucím zkoumání by mohl být rozšířen počet testovaných knihoven a formátů nebo by mohla být testována další platforma. Mezi formáty by mohly být přidány například Apache Thrift, Colfer a další. Z hlediska jazyků by bylo možné zkoumání rozšířit například o C++, C#, Ruby a jiné.

9 Seznam použité literatury

Achour, Mehdi. 2017. serialize,unserialize,Object Serialization. *PHP: Hypertext Preprocessor.* [Online] Červenec 2017. http://php.net/.

Augustýn, Michal. 2017. Proč používat Docker. *Zdroják - o tvorbě webových stránek a aplikací.* [Online] 11. Duben 2017. https://www.zdrojak.cz/clanky/proc-pouzivat-docker/.

Blue, Ryan. 2017. Apache Avro™ 1.8.2 Specification. *Apache Avro.* [Online] 31. Květen 2017. https://avro.apache.org/docs/current/spec.html.

Bray, Tim a Paoli, Jean. 2008. Extensible Markup Language (XML) 1.0 (Fifth Edition). *World Wide Web Consortium (W3C).* [Online] 26. Listopad 2008. https://www.w3.org/TR/REC-xml/.

Bray, Tim. 2014. The JavaScript Object Notation (JSON) Data Interchange Format. *IETF Tools.* [Online] 1. Březen 2014. https://tools.ietf.org/html/rfc7159.

Brown, Paul. 2017. State of the Union: npm. *Linux.com | News for the open source professional.* [Online] 13. Leden 2017. https://www.linux.com/news/event/Nodejs/2016/state-union-npm.

Curbera, Francisco a Duftler, Matthew. 2002. Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. [Online] 7. Srpen 2002. http://tele1.dee.fct.unl.pt/rit2_2009_2010/teo/soap.tutorial.pdf. 1089-7801.

Droettboom, Michael . 2016. Understanding JSON Schema. [Online] 21. Prosinec 2016. https://spacetelescope.github.io/understanding-json-schema/.

DuBuisson, Thomas M. 2017. Native Haskell implementation of Avro. *GaloisInc/avro.* [Online] 13. Červen 2017. https://github.com/GaloisInc/avro.

Dudek, Martin. 2017. Box-Plot neboli Krabicový graf. *Kvalita jednoduše.* [Online] 10. Únor 2017. http://kvalita-jednoduse.cz/box-plot/.

Dvořáková, Veronika. 2016. Nejoblíbenější programovací jazyky. *Buď FIT - Časopis Fakulty informačních technologií ČVUT.* [Online] 17. Březen 2016. https://casopis.fit.cvut.cz/tema/1-2016-nej-programovacich-jazyku/nejoblibenejsi-programovaci-jazyky/.

EL-BAKRY, HAZEM M. a MASTORAKIS, NIKOS. 2009. Studying the Efficiency of XML Web Services for Real-Time Applications. [Online] 2009. http://www.wseas.us/e-library/conferences/2009/baltimore/MAVISE/MAVISE-33.pdf. 978-960-474-135-9.

Elsheimy, Mohammad. 2010. Serialization vs. Marshaling. *C# Corner - A Social Community of Developers and Programmers.* [Online] Čeven 2010. http://www.c-sharpcorner.com/uploadfile/GemingLeader/serialization-vs-marshaling/.

Evans, Clark . 2009. YAML Ain't Markup Language. *YAML.* [Online] 1. Říjen 2009. http://www.yaml.org/.

Fawcett, Joe, Quin, Liam a Danny, Ayers. 2012. *Beginning XML.* Kanada : John Wiley & Sons, Inc., 2012. 978-1-118-16213-2.

Furuhashi, Sadayuki. 2015. MessagePack specification. *msgpack.org.* [Online] 22. Prosinec 2015. https://github.com/msgpack/msgpack/blob/master/spec.md.

Garg, Hitesh. 2014. 7 DIFFERENCES BETWEEN SERIALIZABLE AND EXTERNALIZABLE INTERFACE IN JAVA. *Codingeek A home for Coders.* [Online] 19. Prosinec 2014. http://www.codingeek.com/java/io/differences-serializable-externalizable-interface-java-tutorial/.

Grudl, David. 2017. Neon sandbox. [Online] 2017. https://ne-on.org/.

Harold, Elliotte Rusty. 2003. *Effective XML.* místo neznámé : Addison-Wesley Professional, 2003. 978-0321150400.

Hoff, Arthur. 2010. Java Object Serialization Specification. [Online] 2010. https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html.

Hostetler, Glenn a Hasznos, Sandor. 2009. *Web Service and SOA Technologies.* místo neznámé : Practicing Safe Techs, 2009. 978-0982303702.

Jansen, Paul. 2017. TIOBE Index. *TIOBE Index | TIOBE - The Software Quality Company.* [Online] Červen 2017. https://www.tiobe.com/tiobe-index/.

Kosek, Jiří. 2000. *XML pro každého.* Praha : Grada Publishing, 2000. 80-7169-860-1.

Kotačka, Vít. 2013. Gradle, moderní nástroj na automatizaci. *Zdroják - o tvorbě webových stránek a aplikací.* [Online] 7. Červen 2013. https://www.zdrojak.cz/clanky/gradle-moderni-nastroj-na-automatizaci/.

Kramer, James. 2017. JavaScript. *Mozilla Developer Network.* [Online] 7. Červen 2017. https://developer.mozilla.org/en-US/docs/Web/JavaScript.

Kreps, Jay. 2017. Data Serialization and Evolution. *Confluent: Apache Kafka & Streaming Platform for the Enterprise.* [Online] 2017. http://docs.confluent.io/current/avro.html.

Lynx. 2014. Základy binární serializace. *Hlavní stránka - Ultima Online CZ.* [Online] 20. Srpen 2014. http://ultima.cz/zaklady-binarni-serializace/.

Maeda, Kazuaki. 2011. Comparative Survey of Object Serialization Techniques and the Programming Supports. *World Academy of Science, Engineering and Technology (WASET).* Prosinec 2011, stránky 1488-1493.

Monsch, Matthieu. 2016. Benchmarks. *mtth/avsc.* [Online] 18. Prosinec 2016. https://github.com/mtth/avsc/wiki/Benchmarks.

Newton-King, James. 2015. Converting between JSON and XML. *Json.NET.* [Online] 15. Únor 2015. http://www.newtonsoft.com/json/help/html/ConvertingJSONandXML.htm.

Nurseitov, Nurzhan a Paulson, Michael. 2009. Comparison of JSON and XML data interchange formats: A case study. [Online] Leden 2009. https://pdfs.semanticscholar.org/8432/1e662b24363e032d680901627aa1bfd6088f.pdf.

Pauli, Julien, Ferrara, Anthony a Popov, Nikita. 2013. Serialization. *Table Of Contents PHP Internals Book.* [Online] 2013. http://www.phpinternalsbook.com/classes_objects/serialization.html.

Pragmateek. 2013 . JSON vs. XML: Some hard numbers about verbosity. *CodeProject - For those who code.* [Online] 10. Červen 2013 . https://www.codeproject.com/Articles/604720/JSON-vs-XML-Some-hard-numbers-about-verbosity.

Renaud, Fabien. 2017. Performance testing of serialization and deserialization of Java JSON libraries. *fabienrenaud/java-json-benchmark.* [Online] 21. Květen 2017. https://github.com/fabienrenaud/java-json-benchmark.

Richardson, Leonard a Ruby, Sam. 2007. *Restful Web Services.* Farnham : O'Reilly Media, 2007. 978-0596529260.

Ryan, V., Seligman, S. a Lee, R. 1999. Schema for Representing Java(tm) Objects in an LDAP Directory. *The Internet Engineering Task Force (IETF®).* [Online] Říjen 1999. http://www.ietf.org/rfc/rfc2713.txt.

Sági-Kazár, Márk. 2017. PHP Serialization Benchmarks. *sagikazarmark/php-serialization-bench.* [Online] 13. Leden 2017. https://github.com/sagikazarmark/php-serialization-bench.

Saloranta, Tatu. 2017. Benchmark comparing serialization libraries on the JVM. *eishay/jvm-serializers.* [Online] 7. Květen 2017. https://github.com/eishay/jvm-serializers.

Shabanov, Alexander. 2014. Serialization Benchmark. *avshabanov/perftest-serialization.* [Online] 6. Duben 2014. https://github.com/avshabanov/perftest-serialization.

Shipilev, Aleksey. 2013. The art of Java benchmarking. *Aleksey Shipilëv: one-stop page.* [Online] Listopad 2013. https://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf.

Skeet, Jon. 2017. Language Guide (proto3). *Protocol Buffers.* [Online] 4. Květen 2017. https://developers.google.com/protocol-buffers/docs/proto3.

Strassner, Tom. 2015. XML vs JSON. *School of Engineering - Tufts University.* [Online] 13. Květen 2015. http://www.cs.tufts.edu/comp/150IDS/final_papers/tstras01.1/FinalReport/FinalReport.html#r6.

Suarez, Trevor. 2016. Benchmarking BSON, JSON, and Native Serializing in PHP. *A community of great programmers and their programming tips.* [Online] 25. Únor 2016. https://coderwall.com/p/ccdryg/benchmarking-bson-json-and-native-serializing-in-php.

Surbhi a Chawla, Rama. 2013. Object Serialization Formats and Techniques a Review. *Global Journal of Computer Science and Technology.* 4. Červen 2013, stránky 47-49.

Tiller, Craig. 2007. gRPC - An RPC library and framework. [Online] 15. Červenec 2007. https://grpc.io/.

Tocker , Morgan. 2010. When should you store serialized objects in the database? *Percona – The Database Performance Experts.* [Online] 21. Leden 2010. https://www.percona.com/blog/2010/01/21/when-should-you-store-serialized-objects-in-the-database/.

Trautmann, Tyson. 2012. Debunking the Myths of RPC & REST. *Ethereal Bits.* [Online] 4. Prosinec 2012. http://etherealbits.com/2012/12/debunking-the-myths-of-rpc-rest/.

Tynan, Dan. 2005. Think thin. *InfoWorld - Technology insight for the enterprise.* [Online] 14. Červenec 2005. http://www.infoworld.com/article/2670875/computer-hardware/think-thin.html.

Vieux, Victor. 2017. Get Started, Part 1: Orientation and Setup | Docker Documentation. *Docker Documentation.* [Online] 10. Červen 2017. https://docs.docker.com/get-started/.

Vigh, Christian. 2017. PHP Performance Comparison. *PHP Classes.* [Online] 10. Duben 2017. https://www.phpclasses.org/blog/post/493-php-performance-evolution.html.

Wang, Guanhua. 2011. Improving Data Transmission in Web Applications via the Translation between XML and JSON. *2011 Third International Conference on Communications and Mobile Computing.* 20. Duben 2011.

10 Přílohy

1) Postup instalace
2) Uživatelská příručka
3) Přiložené CD

10.1 Postup instalace

Na přiloženém CD ve složce *benchmark* jsou zdrojové kódy k benchmarkům. Upřednostňovaným operačním systémem je Linux, pro který jsou napsány skripty, které usnadňují následné spuštění. Jedinou závislostí je Docker. Volitelnou závislostí pro Linux je navíc nástroj GNUPLOT, který se stará o vykreslení grafů. Benchmarky lze spustit i bez Docker nebo i na jiném operačním systému. Nicméně preferovaným způsobem je právě Linux v kombinaci s Dockerem, pro které platí následují postup.

1. Pro instalaci Docker postupujte podle návodu na oficiálních stránkách https://www.docker.com.
2. Volitelně lze nainstalovat nástroj GNUPLOT v2. Pro linuxové distribuce Debian nebo Ubuntu stačí spustit příkaz: *apt-get install gnuplot2*
3. Zkopírujte složku *benchmark* z přiloženého CD na svůj systém.
4. Přemístěte se do zkopírované složky.
5. Přidejte spouštěcí práva souborům *run-benchmarks.sh* a *build-docker.sh* (*chmod +x build-docker.sh run-benchmarks.sh*).
6. Spusťte skript *build-docker.sh*, který sestaví docker obrazy, lze spustit s parametrem *-l* a jedním z argumentů *php, java, nodejs,* poté je sestaven obraz jen pro určitou technologii.
7. Benchmark je připraven ke spuštění, *run-benchmarks.sh.*

10.2 Uživatelská příručka

Pro snadnější ovládání benchmarků jsou vytvořeny skripty pro Shell Linux. Jedná se o soubory končící příponou *sh*, aby bylo možné tyto skripty spouštět, je třeba jim přidat spouštěcí právo *x,* viz kapitolu výše. Všechny skripty lze spustit s parametrem *-h* nebo *--help*, který vypíše nápovědu a všechny možné parametry spuštění. Pro správné spuštění skriptů je třeba se nacházet vždy v aktuální složce se skriptem.

10.2.1 Spuštění

Hromadně lze benchmarky spustit pomocí souboru *run-benchmarks.sh* s těmito parametry.

- *-o, --outer <n>* → vnější počet opakování (viz Kód 28).
- *-i, --inner <n>* → vnitřní počet opakování (viz Kód 28).
- *-f, --format <native|json|xml|protobuf|avro|msgpack>* → testuj pouze zadaný formát.
- *-l, --language <php|java|nodejs>* → testuje pouze zvolený jazyk.
- *-c, --chatty* → při testování vypisuje aktuální činnost.

Například tedy *./run-benchmarks.sh -i 1 -o 1 -f msgpack -l php -c.* Tento příkaz spustí benchmarky s jedním vnitřním a jedním vnějším opakováním pouze pro formát msgpack pro jazyk PHP s rozšířeným výpisem.

Výhoda spuštění benchmarku pomocí skriptu *run-benchmarks.sh* je, že jsou navíc z výstupních souborů vytvořeny soubory, které obsahují kombinovaná data všech benchmarků. Pokud je nástroj GNUPLOT nainstalován, jsou navíc z těchto souborů generovány grafy. Ukázka výstupních souborů je na přiloženém CD ve složce *output-example*.

Benchmarky lze spouštět i jednotlivě pomocí skriptů *run-benchmark.sh*, které jsou umístěny v jednotlivých složkách s benchmarky. Tyto skripty navíc přidávají jeden spouštěcí parametr k výše uvedeným, a sice:

- *-r, --result <csv|console>* → formát výstupu.

Při použití argumentu *console* jsou výsledky zobrazeny rovnou do konzole, volba *csv* prezentuje výsledky jako CSV soubory.

Poslední možností spuštění je přímé spuštění bez docker obrazů. V takovém případě je třeba mít nainstalovanou platformu pro spuštění dané technologie. K možnostem spuštění jsou navíc přidány tyto parametry:

- *-t, --data <s>* → cesta k testovacím datům.
- *-d, --out-dir <s>* → cesta k výstupní složce (pokud je formát výstupu nastaven na csv).

Benchmarky lze spouštět pomocí:

- PHP → *php init.php benchmark:run <parametry…>*
- *JAVA → java -jar target/benchmark-java-1.0-jar-with-dependencies.jar <parametry…>*
- JavaScript → *node init.js <parametry…>*

10.2.2 Konfigurace obrazů

Vytvářet lze i vlastní verze docker obrazů, například z důvodů testování na jiné verzi platformy. V takovém případě je nejlepší postupovat podle tohoto návodu.

1. Vytvořit Dockerfile soubor ve složce *benchmark/benchmark-{technologie}/docker/v{verze}*.
2. Dále je nutné editovat skripty pro vytvoření obrazů a spouštění. Editovat lze buď skripty pro hromadné akce, nebo jen skripty pro danou technologii umístěné v příslušných složkách.
    a. V případě konkrétní technologie je třeba editovat soubory *build-docker.sh* a *run-docker.sh* patřící k dané technologii a změnit hodnotu proměnné *version*, která musí odpovídat názvu složky s Dockerfile souborem (bez znaku *v*).
    b. V případě editace hromadných skriptů *build-docker.sh* a *run-dockers.sh* je třeba změnit hodnoty proměnných *versionPHP*

popřípadě *versionNodeJS* nebo *versionJAVA,* tak aby odpovídaly názvům složek s novými Dockerfile soubory (bez znaku *v*).

3. Spustit skript pro sestavení.
4. Spustit benchmark.

10.2.3 Přidání knihovny

Přidání knihovny je obdobné pro všechny tři technologie. V případě PHP je nutné, aby třída s danou funkcionalitou implementovala rozhraní *Benchmark\Metrics\IMetric*. V takovém případě bude zařazena do testování. Rozhraní *IMetric* obsahuje dvě metody *getInfo()* a *run().* První zmíněna metoda musí vrátit objekt typu *Benchmark\Metrics\Info* a druhá metoda objekt typu *Benchmark\Metrics\MetricResult*. Pro snadnější implementaci je připravena abstraktní třída *Benchmark\Metrics\AMetric*, která implementuje metodu *run()* základním způsobem. Přidává několik metod, které jsou postupně volány a hlavně dvě abstraktní metody *serialize()* a *deserialize()*. V těchto metodách by měla být definována funkčnost dané knihovny a právě tyto metody jsou měřeny. Metody jsou volány v tomto pořadí:

1. *prepareBenchmark()* → vhodné pro inicializaci proměnných.
2. *prepareDataForSerialize()* → příprava dat pro serializaci.
3. *serialize()* → serializace dat, měřená metoda.
4. *prepareDataForDeserialize()* → příprava dat pro deserializaci.
5. *deserialize()* → deserializace dat, měřená metoda.

V případě JAVY je postup totožný. Názvy tříd a rozhraní jsou stejné s rozdílem názvu balíčku, který pro JAVU je *benchmark.java*.

Pro JavaScript je třeba vytvořit soubor ve složce *benchmark/benchmark-nodejs/modules/metrics/{formát}/{soubor}.js*, tento soubor musí exportovat objekt vytvořený pomocí konstrukční funkce *metric* ze souboru *benchmark/benchmark-nodejs/modules/metrics/metric.js.* Parametry funkce jsou následující:

1. *format* → jméno formátu (*avro, json, msgpack, protobuf, xml).*

2. *name* → jméno knihovny.
3. *version* → verze knihovny.
4. *url* → webová adresa projektu.
5. *serialize* → funkce pro serializaci.
6. *deserialize* → funkce pro deserializaci.

V případě potřeby je možné přetížit některé metody objektu jako:

1. *metric.prototype.prepareDataForSerialize* → příprava dat pro serializaci.
2. *metric.prototype.serializeImpl* → serializace dat, měřená metoda.
3. *metric.prototype.prepareDataForDeserialize* → příprava dat pro deserializaci.
4. *metric.prototype.deserializeImpl* → deserializace dat, měřená metoda.

10.3 Přiložené CD

Obsah CD:

```
├── benchmark
│   ├── benchmark-java
│   │   ├── docker
│   │   ├── src
│   │   ├── target
│   │   ├── testdata
│   │   ├── build-docker.sh
│   │   ├── pom.xml
│   │   └── run-docker.sh
│   ├── benchmark-nodejs
│   │   ├── docker
│   │   ├── modules
│   │   ├── testdata
│   │   ├── build-docker.sh
│   │   ├── init.js
│   │   ├── package.json
│   │   └── run-docker.sh
│   ├── benchmark-php
│   │   ├── app
│   │   ├── docker
│   │   ├── libs
│   │   ├── temp
│   │   ├── testdata
│   │   ├── vendor
│   │   ├── build-docker.sh
│   │   ├── composer.json
│   │   ├── init.php
│   │   └── run-docker.sh
│   ├── build-docker.sh
│   ├── readme.md
│   └── run-docker.sh
└── example-output
```

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2015/2016

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Bc. Vaňura Jan	Jiskrova 1072, Úpice	I1480

TÉMA ČESKY:

Porovnání formátů pro serializaci dat

TÉMA ANGLICKY:

Comparison of data serialization formats

VEDOUCÍ PRÁCE:

Ing. Pavel Kříž, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Cíl: Porovnat formáty využívané pro serializaci dat (typicky u RESTful webových služeb): především JSON, XML, Message Pack, Protocol Buffers. Vyhodnotit kapacitní a časovou náročnost serializace v různých implementacích (PHP, Java, Node.JS).

Osnova:
1. Úvod
2. Představení formátů
2.1. JSON
2.2. XML
2.3. Message Pack
2.4. Protocol Buffers
3. Dostupné implementace
3.1. PHP
3.2. JAVA
3.3. NODE.js
4. Metodika testování
5. Vyhodnocení testů
6. Závěr

SEZNAM DOPORUČENÉ LITERATURY:

https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=42626
https://pdfs.semanticscholar.org/8432/1e662b24363e032d680901627aa1bfd6088f.pdf

Podpis studenta: ..... **Datum:** .....

Podpis vedoucího práce: ..... **Datum:** .....