UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Volání nativního kódu z javovské aplikace pomocí JNI

Jan Žampach

Bakalářská práce
2011

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2010/2011**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Jan ŽAMPACH**
Osobní číslo:	**I08207**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Volání nativního kódu z javovské aplikace pomocí JNI**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování:

Cílem bakalářské práce bude seznámení s rozhraním Java Native Interface a praktická ukázka jeho implementace.
Teoretická část:
V teoretické části práce bude představeno rozhraní Java Native Interface (JNI), tedy především možnosti jeho využití, role, nevýhody, předávání dat a spolupráce s JVM. Bude popsána práce s nativními dynamickými knihovnami a jejich hlavičkovými soubory.
Implementační část:
U konkrétní vybrané implementace bude provedena analýza problému a navrženo a realizováno vhodné řešení. Vytvořená GUI aplikace v Javě umožní pomocí dostupné dll knihovny, rozhraní JNI a externě připojeného teploměru s A/D převodníkem načítat data, zpracovávat je, vykreslit graf dle zadaných parametrů a exportovat tato data do souboru k dalšímu zpracování.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

*** LIANG, Sheng. The Java Native Interface : Programmer's Guide and Specification [PDF online]. Addison-Wesley, 1999-06, rev. 2002-02-21, [cit. 2009-05-17]. [java.sun.com/docs/books/jni/download/jni.pdf Dostupné online.] ISBN 0-201-32577-2**

*** SIERRA, Kathy ; BATES, Bert . Head first Java. 2nd edition. [s.l.] : O'Reilly, 2005. 688 s. ISBN 978-0-596-00920-5.**

Vedoucí bakalářské práce: **Ing. Zdeněk Šilar**
Katedra informačních technologií

Datum zadání bakalářské práce: **17. prosince 2010**

Termín odevzdání bakalářské práce: **13. května 2011**

Univerzita Pardubice Česká republika

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

Ing. Lukáš Čegan, Ph.D.
vedoucí katedry

V Pardubicích dne 31. března 2011

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 2. 5. 2011 Jan Žampach

Poděkování

Rád bych touto cestou poděkoval panu Ing. Zdeňku Šilarovi za jeho vynaložený čas a ochotnou spolupráci při tvorbě této práce.

Anotace

Práce se věnuje možnostem využití rozhraní JNI při vývoji aplikací, shrnuje výhody a nevýhody použití. Hodnotí JNI z hlediska rychlosti, použitelnosti, přenositelnosti a bezpečnosti oproti jiným technikám programování. V práci je popsán postup při tvorbě aplikace, která využívá nativní dynamickou knihovnu. Na závěr práce je zhodnoceno další možné využití JNI v praxi.

Klíčová slova

JNI, JVM, Java, nativní, dynamická knihovna

Title

Native code call from Java application using JNI

Annotation

The work deals with the possibility of using JNI interface in applications development, summarizes the advantages and disadvantages of implementation. Evaluates JNI in terms of speed, usability, portability and security compared to other programming techniques. The practice part describes how to create an application that uses a native dynamic library. There is other potential of use JNI in practice, reviewed at the conclusion of work.

Keywords

JNI, JVM, Java, native, dynamic library

Obsah

Seznam zkratek

ASM	Assembler
cl	C++ kompilátor programu MS Visual Studio
DLL	Dynamically linked libraries
DOM	Document Object Model
FFI	Foreign Function Interface
g++	GNU kolekce kompilátorů
GC	Garbage Collector
GNU/GPL	GNU General Public Licence
JDOM	Java-based document object model
JIT	Just in Time
JNA	Java Native Access
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
SAX	Simple API for XML
StAX	The Streaming API for XML
USB	Universal Serial Bus
XML	Extensible Markup Language

Seznam obrázků

Seznam tabulek

Seznam grafů

1 Úvod

Tato bakalářská práce si klade za cíl seznámit čtenáře s hlavními aspekty, se kterými bude konfrontován programátor aplikace využívající rozhraní Java Native Interface (JNI), které je součástí platformy Java. Programovací jazyk Java získal za dobu své existence již velmi početnou řadu příznivců. Zasloužil si to především proto, že poskytuje vývojářům nejen programovací jazyk, ale celou platformu – pro mobilní telefony (Java ME), desktopy (Java SE) a distribuované systémy a internetové aplikace (Java EE). Java si kladla od počátku za cíl snadné naučení, pochopení a používání jazyka programátory. To vedlo k eliminaci některých typických konstrukcí jazyků C/C++ – byla zavedena silná typová kontrola, odstraněna vícenásobná dědičnost a přestalo se používat ukazatelů. Java používá pro správu paměti svůj virtuální stroj JVM a přidělování či uvolňování paměti provádí automaticky pomocí tzv. Garbage Collectoru, není tedy potřeba dealokovat instance objektů. Zdrojový kód je kompilovaný do mezikódu (bytecode). Ten je při spuštění buď interpretovaný prostřednictvím JVM, nebo (za účelem dosažení vyššího výkonu) kompilovaný tzv. JIT kompilátorem, tedy za běhu programu do nativního kódu. Mezikód však z principu nemůže dosahovat rychlosti čistě nativního kódu. [1]

I přes to, že Java disponuje velmi širokou platformou knihoven, je někdy potřeba využít napojení na nativní[1] knihovny systému nebo na nativní ovladače periferních zařízení, či využít výkon, který může poskytnout pouze optimalizovaný nativní kód [2]. Tyto případy jsou řešitelné díky rozhraní JNI. Nese to sebou ale nějaká omezení a nevýhody, na které bude v práci poukázáno. Dále bude v práci zhodnoceno možné použití JNI při kombinované implementaci nativního a Java kódu. Za tímto účelem budou provedeny experimenty a budou porovnány rychlosti provádění jednotlivých výpočetně náročných algoritmů. V praktické části bude implementována vybraná komerční dynamická knihovna do Java aplikace.

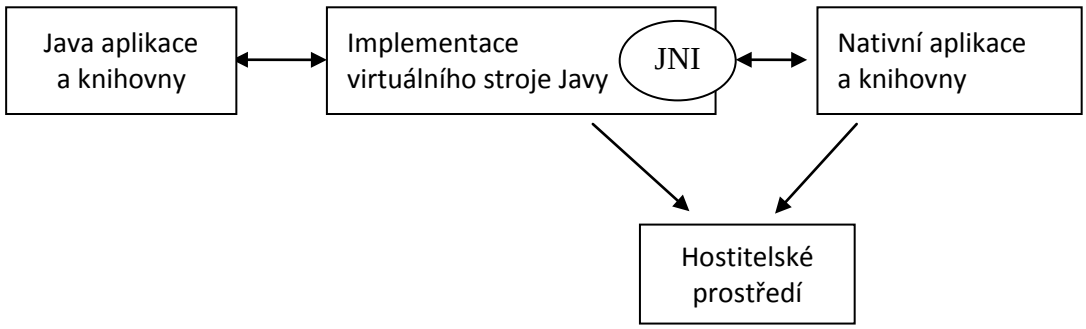
[1] Nativní, neboli přirozený pro konkrétní platformu a OS.

2 Role a využití JNI

Platforma JRE je velice rozsáhlá a poskytuje programátorům velké množství knihoven Java Class Library, které jsou zaručeně součástí každé instalace JRE. Díky této a dalším výhodám se Java často stává platformou nasazovanou k vývoji aplikací na vrcholu hostitelského prostředí. I v těchto případech se může stát, že aplikace vyvíjené v Javě budou nezbytně nutně muset spolupracovat s nativním kódem hostitelského prostředí. V této chvíli nastupuje role JNI. [3, kap. 1.2, 2]

2.1 Pozice JNI při vývoji aplikace

JNI je výkonná funkce či rozhraní, které umožňuje využívat Java platformu, ale nadále i kód napsaný v jiných jazycích. V rámci virtuálního stroje Javy JVM je JNI obousměrným rozhraním, které umožňuje Java aplikaci volat nativní kód a naopak z nativního kódu přistupovat k JVM (obrázek 1) – funguje jako obousměrný interface. JNI podporuje dva typy nativního kódu, a to nativní aplikace a nativní knihovny. [3, kap. 1.2]



Obrázek 1 - Role JNI [3, str. 5]

2.2 Využití JNI

Existují dva základní způsoby, jak přistupovat k JNI a jeho funkcím. [3, kap. 1.2]

a) JNI lze využít při psaní nativních metod, které poskytují Java programům přístup k funkcím implementovaným v nativních dynamických knihovnách. Java program pak tyto metody volá stejným způsobem, jako by byly implementovány v programovacím jazyce Java. Ve skutečnosti jsou ale nativní metody implementovány v jiném programovacím jazyce (typicky C/C++ nebo ASM) a nacházejí se v nativních knihovnách. Jedná se o tzv. přístup shora.
b) JNI podporuje vyvolání rozhraní, které umožní vložit implementaci JVM do nativní aplikace. Jedná se o „opačnou filozofii“ využití, než tomu bylo v minulém případě, tzv. přístup zdola. Nativní aplikace se mohou propojit s nativními knihovnami, které implementují JVM a poté využít tohoto rozhraní ke spuštění softwarových komponent implementovaných v jazyce Java. Typickým příkladem může být webový prohlížeč implementovaný v jazyce C++, který spustí stažený Java applet ve vestavěném JVM.

2.3 Důsledky použití JNI

Hlavní předností jazyka Java je jeho platformní nezávislost. Tu ale bohužel program používající JNI do jisté míry ztrácí. Nativní knihovny a aplikace, se kterými javovská aplikace komunikuje, jsou totiž závislé na platformě. Při snaze zachovat přenositelnost je tedy nutné tuto nativní část kompilovat pro každou platformu zvlášť. Na straně javovské aplikace pak lze ošetřit tento případ následujícím způsobem.

```
System.loadLibrary(Platform.isWindows() ? "win_lib" : "unix_lib");
```

Tedy nativní část aplikace je zkompilovaná pro prostředí Windows a UNIX zvlášť. V závislosti na hostitelském prostředí se načte správná knihovna (např. v případě Windows .dll, v případě UNIXu .so).

Dalším problémem, na který by měl programátor dát při používání JNI pozor, je částečná ztráta typové bezpečnosti. Nativní jazyky C nebo ASM silně typově bezpečné nejsou. V C lze např. přiřadit proměnnou jednoho datového typu do jiného datového typu. Z toho důvodu je nutné dbát zvýšené opatrnosti při kontrole typové bezpečnosti Java kódu před voláním nativních metod.

Z odborné literatury plyne [3, str. 7], že základním pravidlem při navrhování Java aplikace, která využívá rozhraní a funkcí JNI, je snaha přesunout co největší část implementovaného kódu do části Java aplikace a v nativním kódu implementovat jen holé, nezbytně nutné minimum kódu. Nativní část aplikace se totiž velmi špatně ladí. Omezené je nasazení debuggeru na nativní část, protože ta je kompilována do dynamické knihovny. Veškeré výjimky a nastalé nestandardní situace se vyplatí implementovat až na „vyšší úrovni“, tedy v Java kódu, spolu se zbylou částí implementovaného kódu.

2.4 Postup při implementaci

V následujících podkapitolách bude na jednoduchém příkladu demonstrována filozofie práce s JNI rozhraním, detailněji popsána v [3, kap. 2, 3]. Jako příklad byl zvolen výpočet n-tého čísla Fibonacciho posloupnosti.

2.4.1 Vytvoření třídy s deklarací nativní metody

Java třída `Test` obsahuje nejprve deklaraci statické nativní metody `fibonacci`, dále metodu `main`, ve které je metoda `fibonacci` zavolána. Za metodou `main` se nachází statický iniciátor, který načte knihovnu s implementací nativní metody.

```
public class Test {

    private native static long fibonacci(int n);

    public static void main(String[] args) {
        int n = 40;
        long vysledek = fibonacci(n);
        System.out.println(n + ". číslo fibonacciho posloupnosti je " +
vysledek);
    }
    static {
        System.loadLibrary("Test");
    }
}
```

Klíčové slovo `native` odlišuje metodu `fibonacci` od klasické metody v programovacím jazyce Java. Znamená, že tělo metody je implementováno v jiném programovacím jazyce. S tím souvisí i ukončení metody středníkem – jedná se pouze o deklaraci. Nativní implementace se nebude nacházet zde, nýbrž až v separátním .cpp souboru.

Dříve, než bude možné zavolat nativní metodu `fibonacci`, musí být načtena do paměti knihovna, ve které se nachází její implementace. O to se postará tzv. statický iniciační blok[2], díky kterému je garantováno, že knihovna bude včas načtena do paměti – tj. před jejím zavoláním. V programovacím jazyce Java je načítání knihoven zajištěno třídou `System`, metodou `loadLibrary()`. Parametrem je jméno knihovny (bez přípony), která se má načíst. Načítaná knihovna se musí nacházet ve vyhledávací cestě `Java.library.path`.

2.4.2 Kompilování Java třídy

Po uložení souboru `Test.java` se zdrojovým kódem, je nutné zkompilovat soubor do bytekódu kompilátorem `javac`.

```
javac Test.java
```

Příkaz `javac` zkompiluje soubor do stejného adresáře s příponou `class`.

[2] Více informací např. na http://download.oracle.com/javase/tutorial/java/javaOO/initial.html

2.4.3 Hlavičkový soubor nativní funkce

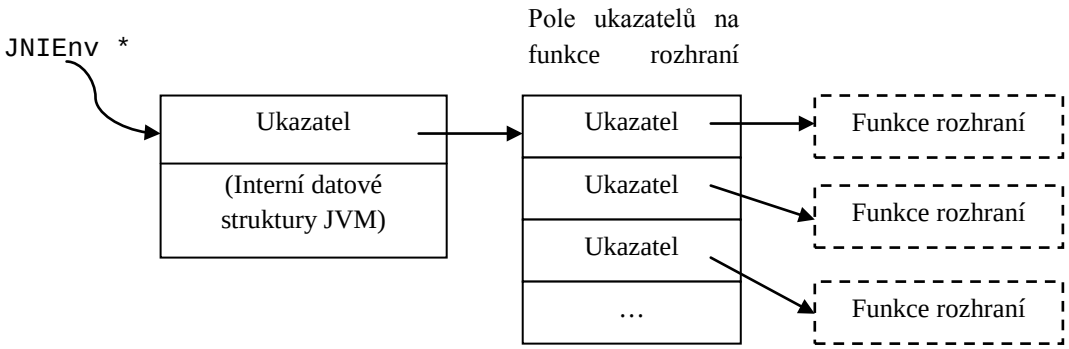
Pro korektní implementaci nativní funkce je nyní nutné nechat si vygenerovat její hlavičku. K tomu slouží program `javah`.

```
javah -jni Test
```

Soubor vygenerovaný programem `javah` bude mít název `Test.h` a bude obsahovat prototyp funkce `fibonacci`.

```
/*
 * Class:     Test
 * Method:    fibonacci
 * Signature: (I)J
 */
JNIEXPORT jlong JNICALL Java_Test_fibonacci (JNIEnv *, jclass, jint);
```

Makra `JNIEXPORT` a `JNICALL` ujišťují, že se jedná o funkce exportované z nativní knihovny a C/C++ kompilátor vygeneruje kód se správnou konvencí pro tuto funkci. Mezi makry se nachází návratový typ `jlong`, viz kapitola (2.5.1). Za makrem `JNICALL` se nachází jméno funkce, které se skládá z několika části, a to prefixu `Java_`, následuje jméno třídy v Javě, tedy `_Test` a konečně název metody `_fibonacci`. Deklarace metody `fibonacci` v Javě měla jen jeden argument, a to n-té číslo posloupnosti. Nicméně v kódu výše je vidět, že vygenerovaný funkční prototyp má argumenty tři. Prvním argumentem je ukazatel na rozhraní `JNIEnv`, ukazující na místo, které obsahuje ukazatel na tabulku funkcí, viz obrázek 2. Každý zápis v tabulce ukazuje na funkci rozhraní JNI. Nativní funkce se tedy vždy odkazují na datové struktury v JVM skrz danou funkci JNI.



Obrázek 2 - Ukazatelé na funkce rozhraní [3, str. 23]

Druhým argumentem (`jclass`) je reference na třídu `Test`. Nativní metoda je statická, proto se vyskytuje v argumentu reference na třídu. Pokud by však metoda byla nestatická, tedy instanční, program `javah` by do hlavičky vygeneroval referenci na objekt (`jobject`), kterým je metoda iniciována. Argumenty `JNIEnv` a `jclass` (potažmo `jobject`) bude mít každá funkce vygenerovaná programem `javah`.

2.4.4 Implementace nativní funkce

Díky programu `javah` je zajištěn korektní předpis funkčního prototypu, který je nutný v definici funkce dodržet. Následující kód ukazuje implementaci nativní funkce v jazyce C++. Ve funkci není využita žádná funkce z rozhraní JNI.

```
#include <jni.h>
#include "Test.h"
/*
 * Class:     Test
 * Method:    fibonacci
 * Signature: (I)J
 */
JNIEXPORT jlong JNICALL Java_Test_fibonacci
(JNIEnv *env, jclass obj, jint n)
{
    if (n < 2)
        return (1);
    else
        return (fib(n - 2) + fib(n - 1));
}
```

Samotná implementace v jazyce C++ je jednoduchá a jasná. K tomu, aby bylo možné později tento zdrojový kód zkompilovat, je nutné přidat direktivou `include` soubor `jni.h`, který poskytuje nativnímu kódu možnost přistupovat k funkcím rozhraní JNI. Tento soubor je nutné vkládat vždy.

2.4.5 Kompilace do nativní knihovny

Ze souboru `Test.cpp` je nyní potřeba vytvořit sdílenou knihovnu, která bude posléze načtena ve statickém iniciačním bloku v Java části programu. Knihovnu je možné vytvořit na různých platformách různými způsoby, např. ve vývojových prostředích typu NetBeans nebo MS Visual Studio. Dalším způsobem je možnost využít kompilátor přímo v příkazové řádce s danými parametry. Na platformě Windows lze zkompilovat zdrojový kód do dynamické knihovny např. kompilátorem *cl* (MS Visual Studio) tímto způsobem:

```
cl -MD –LD -I"C:\Java\jdk1.6.0_18\include"
-I"C:\Java\jdk1.6.0_18\include\win32" Test.cpp -FeTest.dll
```

Přepínač `-MD` zajišťuje přilinkování knihovny `msvcrt.lib`, nutné k fungování dynamické knihovny, přepínač `-LD` určuje typ výstupního souboru, tedy dll a ne spustitelný binární soubor. Přepínač `-I` značí, že za ním budou následovat cesty k vkládaným souborům direktivou `include`. Podobným způsobem lze zkompilovat i dynamickou knihovnu pro systémy typu UNIX, např. s použitím kompilátoru *gcc*:

```
gcc -G -I/java/include -I/java/include/unix Test.cpp -o Test.so
```

V tomto případě značí přepínač `-G` typ výstupního souboru, tedy sdílenou knihovnu.

2.4.6 Spuštění programu

Všechny potřebné kroky k vytvoření Java programu, který volá nativní metodu, byly splněny. Program lze tedy spustit např. z příkazové řádky.

```
java Test
```

Ke správnému spuštění je bezpodmínečně nutné ještě umístit sdílenou knihovnu do správného adresáře, resp. mít správně nastavené vyhledávací cesty v proměnné systému PATH. Pokud by se programu nepodařilo správně načíst sdílenou knihovnu, vrátil by výjimku `java.lang.UnsatisfiedLinkError`.

2.5 Práce s datovými typy

Vzájemné předávání dat mezi nativním prostředím a prostředím JVM je zajištěno díky množině definic C/C++ typů, které definuje JNI (v souboru jni.h) a které korespondují s typy v programovacím jazyce Java, kde lze rozlišit dva druhy typů, a to primitivní a referenční typy.

2.5.1 Primitivní typy

Primitivní datové typy v programovacím jazyce Java souvisí s typy, které definuje JNI. Jejich název, velikost a související typ v Javě ukazuje tabulka 1.

Tabulka 1 – Korespondující primitivní typy v Javě, C/C++ a JNI [5]

Typ v Javě	Typ v JNI	Typ v C/C++	Poznámka
`boolean`	`jboolean`	`unsigned char (bool)`	Pro kompatibilitu s C není použit *bool* z C++.
`byte`	`jbyte`	`unsigned char`	
`char`	`jchar`	`unsigned short (wchar_t)`	Java pro znaky používá kódování UTF-16 (dříve UCS-2).
`short`	`jshort`	`short (int16_t)`	
`int`	`jint`	`int (int32_t)`	Starší C kompilátory měly *int* 16bitový.
`long`	`jlong`	`long long (int64_t)`	Velikost *long* se v C typicky liší podle velikosti ukazatele na platformě, proto *long long*.
`float`	`jfloat`	`float`	32bitové desetinné číslo
`double`	`jdouble`	`double`	64bitové desetinné číslo

Tyto typy jsou definovány direktivou typedef v hlavičkovém souboru jni.h.

```
typedef unsigned char jboolean;
```

2.5.2 Referenční typy

Práce s referenčními typy jazyka Java je v nativním prostředí zabezpečena díky funkcím JNI pro práci s nimi. JNI definuje referenční typy, které shrnuje tabulka 2. Referenční typy jsou, stejně jako primitivní typy, definovány v hlavičkovém souboru jni.h.

Tabulka 2 – Korespondující referenční typy v JNI a Javě [3, str. 166]

Typ v JNI	Typ v Javě
`jclass`	`java.lang.Class` instance
`jstring`	`java.lang.String` instance
`jobjectArray`	`Object[]`
`jbooleanArray`	`boolean[]`
`jbyteArray`	`byte[]`
`jcharArray`	`char[]`
`jshortArray`	`short[]`
`jintArray`	`int[]`
`jlongArray`	`long[]`
`jfloatArray`	`float[]`
`jdoubleArray`	`double[]`
`jthrowable`	`java.lang.Throwable` objekty

Některé referenční typy dědí od svých předků, např. `jclass`, `jthrowable`, `jstring` a `jarray` dědí od třídy `jobject`. Všechny referenční typy pro práci s poli jsou odvozené od třídy `jarray`.

```
class _jintArray : public _jarray {};
typedef _jintArray *jintArray;
```

2.6 Práce s třídou String

S objekty třídy String se v nativním kódu pracuje jako s referenčním typem `jstring`, jak bylo popsáno výše. Nelze s ním tedy v nativním kódu pracovat jako s ukazatelem na pole znaků, jako to lze v C/C++. Lze toho však dosáhnout s využitím konverzních funkcí, které poskytuje rozhraní JNI. Tyto podporují jak řetězce kódované v Unicode, tak i v UTF-8 (pro každé kódování jiná funkce). Pro konverzi řetězců do nativního typu, tedy přesněji k získání ukazatele na pole znaků slouží funkce

```
const jbyte * GetStringUTFChars(JNIEnv *env, jstring string,
jboolean *isCopy);
```

Po získání ukazatele lze již pracovat s řetězcem klasicky jako s C polem. Pokud je návratová hodnota `NULL`, nepodařilo se alokovat paměť a zároveň je vrácena výjimka `OutOfMemoryError`. Po ukončení práce s polem je nutné uvolnit přidělenou paměť JNI funkcí:

```
void ReleaseStringUTFChars(JNIEnv *env, jstring string, const char *utf);
```

Podobným způsobem je možné alokovat nový `jstring`, tedy novou instanci třídy `java.lang.String` voláním JNI funkce:

```
jstring NewStringUTF(JNIEnv *env, const char *bytes);
```

Ta z předaného C pole znaků vytvoří novou instanci třídy `String`, reprezentující stejnou sekvenci Unicode znaků jako předané pole UTF-8 znaků. Funkce také vrací výjimku `OutOfMemoryError`, pokud se JVM nepodaří alokovat paměť pro nově vzniklou instanci. Další funkce JNI pro práci s řetězci jsou popsány v [3, kap. 3.2].

2.7 Práce s poli

Způsob práce s poli v souvislost s JNI je odlišná pro pole primitivních a referenčních typů. Pro pole složená z primitivních typů platí podobná filozofie práce jako pro práci s řetězci.

2.7.1 Pole primitivních datových typů

Pro práci s poli primitivních datových typů nabízí rozhraní JNI množinu funkcí typu `Get/Set/Release`, pomocí kterých lze plnohodnotně s poli zacházet. Např. funkce

```
jint * GetIntArrayElements(JNIEnv *env, jintArray arr, jboolean *isCopy);
```

efektivně poskytne ukazatel přímo na Java `int` pole, tedy všechny změny provedené v nativním kódu se promítnou i po návratu do JVM. Tato funkce samozřejmě existuje i pro další datové typy, mění se pouze název datového typu (`Get<Typ>ArrayElements`), viz tabulka 3. Jak bylo uvedeno v kapitole výše, filozofie použití dalších funkcí uvedených v tabulce 3, je podobná jako u JNI funkcí pro práci s řetězci.

Tabulka 3 – Vybrané funkce JNI pro práci s primitivními poli [3, str. 30]

JNI Funkce	Popis
`Get<Typ>ArrayRegion` `Set<Typ>ArrayRegion`	Zkopíruje obsah pole do nebo z přealokovaného bufferu
`Get<Typ>ArrayElements` `Release<Typ>ArrayElements`	Získá ukazatel přímo na pole primitivních typů (volitelně získá kopii pole)
`GetArrayLength`	Vrací počet prvků v poli
`New<Typ>Array`	Vytvoří nové pole požadované délky
`GetPrimitiveArrayCritical` `ReleasePrimitiveArrayCritical`	Získá nebo uvolní ukazatel na pole (volitelně zakáže funkci GC)

Funkce `GetArrayLength` vrací počet prvů pole primitivních, ale i referenčních typů. Funkce `Get/Set<Typ>ArrayRegion` kontrolují meze pole a v případě jejich překročení je vyhozena výjimka `ArrayIndexOutOfBoundsException`.

Od verze Javy 2 (vydání 1.2) jsou dostupné i funkce `Get/Release <Typ>PrimitiveArrayCritical`, které umožňují JVM zakázat činnost garbage collectoru ve chvíli, kdy nativní kód přistupuje k obsahu polí primitivních datových typů (podobnými funkcemi jsou `Get/ReleaseStringCritical` týkající se řetězců). V kódu mezi funkcemi `Get/Release<Typ>PrimitiveArrayCritical` nesmí být zavolána

žádná z jiných funkcí JNI nebo nesmí být provedena žádná blokující operace, která by mohla způsobit deadlock. [3, kap. 3.2.7]

2.7.2 Pole referenčních datových typů

Pro přístup k referenčním polím poskytuje JNI speciální pár funkcí. Jsou jimi `GetObjectArrayElement`, která poskytuje prvek pole na dané pozici a `SetObjectArrayElement`, která naopak nastavuje prvek na dané pozici. JNI neposkytuje funkce, které by umožňovaly přístup k celému poli (ukazateli) nebo jeho kopii. Více v [3, kap. 3.3.5].

2.8 Práce s výjimkami

Výjimky slouží ve všech programovacích jazycích jako nástroje k zachycení nestandardních situací. Nejinak tomu je v případě JNI, které poskytuje několik funkcí pro práci s výjimkami.

Většina funkcí JNI vrací v případě chyby návratovou hodnotu `NULL` nebo -1, s kterou lze při zpracování výjimek pracovat, nicméně někdy tato hodnota může být legální návratovou hodnotou funkce a vyplatí se použít jiné, efektivní řešení, které nabízí funkce JNI `ExceptionCheck`. Ta garantovaně vrací hodnotu `JNI_TRUE` v případě, že byla v předcházejícím výkonu programu zachycena výjimka. Tuto je potom nutné zachytit, ošetřit a vymazat ze stavového prostoru JNI. Zároveň je nutné v případě zachycení výjimky uvolnit všechny alokované zdroje. Samotné zachycení výjimky může vypadat následovně:

```
(*env)->GetIntArrayElements(env, arr, NULL);
if ((*env)->ExceptionCheck(env)) {
/* Vyskytla se výjimka při získávání ukazatele na pole */
      jclass newExcCls;
      (*env)->ExceptionDescribe(env);
      (*env)->ExceptionClear(env);
      newExcCls = (*env)->FindClass(env,
"java/lang/IllegalArgumentException");
      if (newExcCls == NULL) {
      /* Nepodařila se najít třída IllegalArgumentException */
            return;
      }
      (*env)->ThrowNew(env, newExcCls, "Chybová zpráva");
      (*env)->DeleteLocalRef(env, newExcCls);
}
...
```

`ExceptionDescribe` funkce popíše vzniklou chybu do stavového prostoru JNI a `ExceptionClear` vzniklou výjimku vymaže z fronty vzniklých výjimek. Pomocí funkce `FindClass` se přiřadí proměnné `jclass` Java třída, reprezentující vzniklou výjimku. Tato je poté funkcí `ThrowNew` vrácena do JVM spolu s chybovou zprávou. JVM poté zachytí a zpracuje výjimku jako klasicky vzniklou v metodě v programovacím jazyce Java. Alokovaná třída typu `jclass` musí být po vyhození výjimky opět dealokována pomocí funkce `DeleteLocalRef`. [3, kap. 6.1, 6.2]

3 Alternativy k JNI

V závislosti na daných okolnostech se mohou nabízet alternativní techniky komunikace mezi nativním a javovským kódem. JNI nachází uplatnění především v případech, kdy jsou všechny části aplikace zapouzdřeny v jednom celku, nicméně mohou nastat situace, kdy programátor může použít i jiné řešení komunikace s nativním kódem.

3.1 Meziprocesová komunikace

Komunikace mezi procesy (IPC) je velmi rozsáhlé téma, zabývající se mnoha způsoby, jak si procesy mohou předávat mezi sebou data. Po omezení na programovací jazyk Java a na možnost podobného využití jako u JNI jsou alternativy následující.

3.1.1 Komunikace prostřednictvím vstupně výstupních proudů

Jedním z možných způsobů, jak lze komunikovat z prostředí Javy s nativními programy, je meziprocesová komunikace realizovaná pomocí tříd `Process`, `ProcessBuilder` a vstupně – výstupních proudů `BufferedReader`.

```
Process p = new ProcessBuilder("proces1", "arg1").start();
```

Proces s názvem `proces1` je spuštěn s parametrem `arg1`. Komunikaci s procesem zajišťuje třída `Process` metodami `getInputStream` a `getOutputStream`. Programy spolu tedy komunikují prostřednictvím svých standardních vstupů a výstupů, přičemž jeden program odesílá na svůj standardní výstup data a druhý program na svém standardním vstupu naslouchá a data přijímá a naopak. [6]

```
inStream = new BufferedReader(new InputStreamReader(p.getInputStream()));
inStream.readLine();
```

3.1.2 Komunikace pomocí socketů

Dalším prostředkem pro meziprocesovou komunikaci a alternativou k JNI jsou sockety. Na rozdíl od jiných komunikačních prostředků jsou určeny i pro komunikaci procesů mezi dvěma a více počítači. Sockety jsou koncové body komunikačního kanálu, kterým proudí data mezi jedním serverem a jedním nebo několika klienty. Socket je po nastavení programem (serverem) svázán s daným hardwarovým portem počítače, kde program běží, tedy na tento port mohou jakékoliv programy (klienti) v síti se sockety asociovanými na stejný port posílat data a komunikovat tak se serverem. Následující ukázka kódu ilustruje získání instancí objektů vstupního a výstupního proudu. [7]

```
try{
   in = new BufferedReader(new InputStreamReader(
                            client.getInputStream()));
   out = new PrintWriter(client.getOutputStream(),
                         true);
  } catch (IOException e) {
    System.out.println("Čtení selhalo.");
    System.exit(-1);
  }
```

Bohužel v případě využití třídy `Process` i socketů je režie na přenos dat přímo úměrná jejich velikosti. Na rozdíl, JNI efektivně sdílí paměť a např. velká pole jsou předávána pouze referencí, takže se nemusí vytvářet žádné kopie a s daty se pracuje přímo.

Řešení pomocí třídy `Process` i socketů v Javě jsou velmi jednoduchá a najdou uplatnění při implementaci platformně závislých operací, nebo při řešení výpočetně náročných výpočtů, které mohou být rychlejší v jazyce C nebo C++, ale nevyžadují velké přenosy dat mezi komunikujícími procesy. Aplikace se pak stává závislá na jiném, odděleném programu.

3.2 Java Native Access

JNA (Java Native Access) je jednodušší a novější varianta k JNI. Poskytuje snadný přístup ke sdíleným dynamickým knihovnám bez psaní jakéhokoli jiného kódu než Javy. Přístup je dynamický, tzn. za běhu programu. JNA dovoluje přímo a přirozeně volat nativní funkce, jako by se jednalo o klasické metody. Knihovna JNA používá malou nativní knihovnu (FFI), pomocí které volá nativní kód. Na programátorovi je tedy jen popsat díky této knihovně nativní funkce a struktury. Díky těmto vlastnostem je poměrně jednoduché využít přirozených vlastností platformy na rozdíl od JNI, kde programátor brát ohledy na platformu musí. JNA klade důraz především na jednoduchost použití a správnost kódu před výkonností. Největším rozdílem JNA od rozhraní JNI je nemožnost přistupovat k JVM ze strany nativního kódu. [8]

```
import com.sun.jna.Library;
import com.sun.jna.Native;
import com.sun.jna.Platform;

/** Simple example of JNA interface mapping and usage. */
public class HelloWorld {

// This is the standard, stable way of mapping, which supports
// extensit customization and mapping of Java to native types.
    public interface CLibrary extends Library {
        CLibrary INSTANCE = (CLibrary)
            Native.loadLibrary((Platform.isWindows() ? "msvcrt" : "c"),
                               CLibrary.class);

        void printf(String format, Object... args);
    }

    public static void main(String[] args) {
        CLibrary.INSTANCE.printf("Hello, World\n");
        for (int i=0;i < args.length;i++) {
            CLibrary.INSTANCE.printf("Argument %d: %s\n", i, args[i]);
        }
    }
}
```

4 Export dat pomocí XML

4.1 Charakteristika jazyka XML

Značkovací jazyk XML (nebo také metajazyk) se vyvinul jako podmnožina z velmi komplexního standardu SGML. XML si zachoval možnost definovat vlastní DTD a vlastní značkovací jazyk pro jednotlivé skupiny dokumentů. XML byl vyvinut primárně pro uchovávání a zpracování textových dokumentů, nicméně značky dokážou zachytit důležité informace o samotné struktuře a významu dokumentu nebo povaze dat. XML dokument se tedy výborně hodí pro výměnu dat mezi aplikacemi a konverze do libovolného formátu je potom relativně snadná. XML je standardizován konsorciem W3C. To znamená, že se jedná o otevřený formát a je tedy snadné implementovat ho na všech platformách. [10]

4.2 XML parsery podporované jazykem Java

4.2.1 DOM

DOM neboli objektový model dokumentu je jazykově a platformně nezávislým rozhraním, které umožňuje dynamický přístup a aktualizaci obsahu, struktury a stylu dokumentu. Je to API nezávislé na programovacím jazyku, usnadňující zpracování dokumentů XML. Je založen na objektové reprezentaci XML dokumentu prostřednictvím stromu. [11]

4.2.2 SAX

SAX je jednoduché API pro přístup k dokumentům XML, původně vyvinuté pouze pro platformu Java. Dnes se stal SAX standardem a je použitelný v mnoha programovacích jazycích. Oproti DOMu je SAX založen na tzv. událostním modelu (nebo také proudovém čtení[3]). Díky této odlišnosti se SAX API stává výhodným a rychlým při operacích s objemnými XML dokumenty, nicméně klade větší nároky na zkušenosti programátora. [12]

4.2.3 StAX

Jedná se taktéž o API založené na podobném modelu jako SAX, ale s tím rozdílem, že StAX API generuje události na žádost, tzn. na pokyn programátora reaguje vrácením příslušného úseku. Poskytuje dvě úrovně přístupu, a to pomocí kurzorů, nebo pomocí objektově založených událostí. [13]

4.2.4 JDOM

Vznikl s cílem skloubit výhody parserů DOM a SAX a vyvinout tak komplexní, intuitivní a optimalizované rozhraní pro práci s XML v programovacím jazyce Java. Dovoluje využít veškeré vlastnosti Javy, jako je přetěžování, reflexe, používání kolekcí apod. Umožňuje práci se SAX proudy i DOM stromy. Jedná se o open-source projekt, dostupný pod licencí LGPL. JDOM existuje v podobě externích knihoven, které se musí do projektu importovat. [14], [15]

[3] Aplikace přijíma informace z XML dokumentu v souvislém proudu.

5 Experimentální porovnání jazyků C++ a Java v konfrontaci s JNI

Cílem tohoto experimentu bylo zjistit rozdíly v rychlostech provádění daných výpočetně náročných algoritmů implementovaných v různých programovacích jazycích. Dále možnost jejich kombinované implementace, např. v programovacím jazyce Java a C++ s využitím JNI. Porovnány byly třídící algoritmy Quick sort, Heap sort a Insertion sort, dále pak rekurzivně implementovaný algoritmus výpočtu n-tého čísla Fibonacciho posloupnosti a konečně algoritmus založený na provádění operací s textovými řetězci, konkrétně napojování.

Tabulka 4 – Konfigurace počítačů použitých při testech[4]

	Konfigurace 1	Konfigurace 2
Operační systém	MS Windows 7 Professional	Ubuntu 10.04 server
Sestavení	7600	2.6.35-22-generic-pae
Procesor	Intel Core2 Duo T8100 @2100 MHz	AMD Sempron 3000+ @1800 MHz
RAM	2048 MB	496 MB

Tabulka 5 – Kompilátory použité při testech[5]

	Kompilátor 1	Kompilátor 2
Název kompilátoru	cl	g++
Licence	MS Visual Studio	GNU/GPL
Použité optimalizace	-Ox	-O3

5.1 Význam dynamických knihoven

Za knihovnu lze obecně označit soubor obsahující program, který může být využíván jiným programem. Tyto knihovny zpravidla dělíme na statické a dynamické. Každá má svou výhodu i nevýhodu. Zásadní rozdíl mezi nimi je ve způsobu jejich použití. Statická knihovna je slinkována s programem, který bude využívat její funkcionalitu v době kompilace tohoto programu. Statické knihovny najdou uplatnění v relativně jednoduchých programech, kde je vyžadováno pouze malé funkcionality. Každá instance programu má svojí vlastní kopii knihovny v paměti. Zatímco dynamická knihovna je programem načtena až v době běhu programu, tedy dynamicky. Uplatní se především tam, kde bude využita několika programy najednou – v paměti se uchovává pouze jedna kopie této knihovny a ta může být současně volána několika programy. Ušetří se tak místo. [9]

[4], [5] Zdroj: vlastní zpracování

5.2 Kompilace dynamické knihovny

Dynamická knihovna byla vytvořena za účelem ověření rychlosti algoritmů implementovaných v nativním kódu, pokud jsou volány z Java Virtual Machine a ověření režie při volání funkcí z dynamické knihovny a předávání dat. Knihovna byla kompilována kompilátorem 1 s parametry uvedenými v tabulce 6.

```
cl -MD -LD -Ox -I"C:\Program Files\Java\jdk1.6.0_18\include"
-I"C:\Program Files\Java\jdk1.6.0_18\include\win32" MainTest.cpp
-FeMainTest.dll
```

Tabulka 6 – Významy jednotlivých přepínačů při kompilaci dynamické knihovny[6]

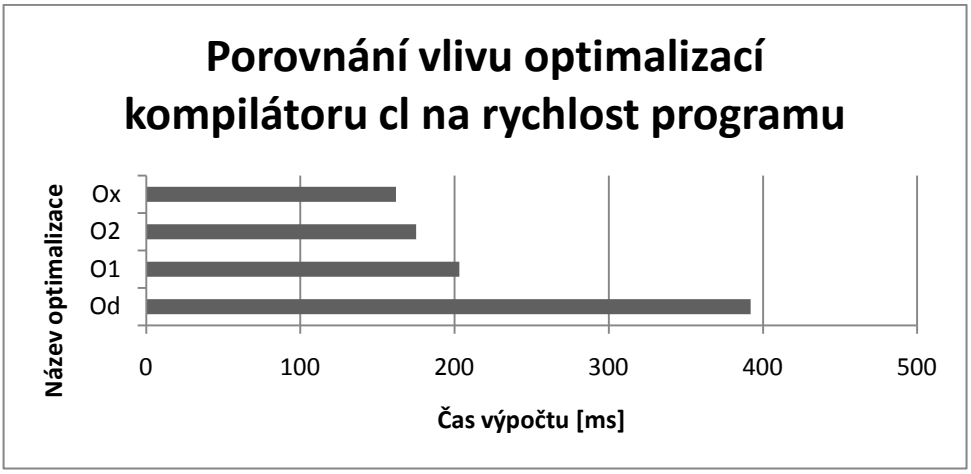
`-MD`	Přilinkování knihovny msvcrt.lib
`-LD`	Určení typu výstupního souboru (.dll)
`-Ox`	Maximální míra optimalizací
`-I`	Určení souborů, které jsou nutné ke kompilaci
`-Fe`	Jméno výstupního souboru

5.3 Optimalizace kódu kompilátorem

Motivace k optimalizaci kódu může být různá. U aplikací náročných na procesorový čas však bývá jasným cílem tento čas minimalizovat. Samotný proces optimalizace by měl začít již při návrhu strategie řešeného problému, dále pak pokračuje výběrem použitých algoritmů, výběrem správného programovacího jazyka a jeho možnostmi a v neposlední řadě také závisí na zkušenostech programátora.

Při porovnávání rychlostí algoritmů byl kladen důraz na optimalizace kompilátorů C++ kódu. Kompilátor 1 i 2 nabízejí několik možných úrovní optimalizací. Každá úroveň optimalizace je specializována na jinou oblast optimalizace. Např. u kompilátoru *cl* lze parametrem *-O1* optimalizovat velikost zkompilovaného kódu, *-O2* rychlost, *-Og* povoluje globální optimalizace kódu, *-Od* zakazuje všechny optimalizace. Porovnání některých optimalizací v závislosti na výsledné rychlosti prováděného programu ukazuje následující graf 1.

[6] Zdroj: nápověda příkazu *cl*



Graf 1 - Porovnání vlivu optimalizací kompilátoru *cl* na rychlost programu

Z grafu 1 je patrné, že optimalizace mají na výslednou rychlost kódu obrovský vliv. Experiment porovnání vlivu různých optimalizací kompilátoru na výslednou rychlost programu byl proveden na algoritmu Quick sort, který měl seřadit 10^6 prvků – náhodně generovaných čísel. Program kompilovaný kompilátorem *cl* s parametrem *-Ox* provedl seřazení stejných prvků 2,4krát rychleji, než program kompilovaný bez optimalizací (*-Od*). K experimentu byla použita konfigurace 1 a kompilátor 1 (Tabulka 4, Tabulka 5).

5.4 Výsledky provedených experimentů

Experimenty ukázaly, že výpočetně náročné algoritmy jsou po optimalizované kompilaci až téměř dvakrát rychleji provedené (v případě kompilátoru *cl*) oproti programům implementovaným v Javě. Naopak v případě rekurzivně implementovaného algoritmu na výpočet n-tého Fibonacciho čísla, byl nejrychleji hotov program kompilovaný kompilátorem 2 na konfiguraci 2, nicméně program v Javě vypočítal 40. číslo Fibonacciho posloupnosti jako druhý v pořadí. Všechny grafy jsou uvedeny v příloze a zdrojové kódy všech experimentálních programů na přiloženém CD.

Tabulka 7 – Výsledky provedených experimentů[7]

		Testovaný algoritmus				
		Quick sort	**Heap sort**	**Insertion sort**	**spojování řetězců**	**Fibonacci**
Použitý kompilátor	**cl**	**162 ms**	234 ms	2896 ms	**43 ms**	3651 ms
	g++ Windows	192 ms	**232 ms**	3776 ms	527 ms	3778 ms
	g++ Linux	268 ms	510 ms	6725 ms	75 ms	**1245 ms**
	Java - jni - dll(cl)	176 ms	272 ms	**2606 ms**	51 ms	3315 ms
	Java	317 ms	376 ms	6020 ms	108 ms	1878 ms
	parametr funkce[8]	10^6	10^6	10^5	10^6	40

[7] Graf 1 až Graf 6, Tabulka 7 – zdroj: vlastní zpracování

[8] Udává, s jakým parametrem byla funkce/metoda volána. U řadících algoritmů udává velikost řazeného pole; u výpočtu Fibonacciho čísla udává, kolikáté číslo posloupnosti je počítáno; u spojování řetězců znamená, kolik řetězců bylo spojeno.

5.5 Poznámky k provedeným experimentům

Ke generování náhodných čísel byla použita vlastní třída generátoru, který byl naprogramován na principu lineárního kongruentního generátoru. Byl implementován v Javě i v C++ stejným způsobem tak, aby při spuštění generoval vždy od stejné počáteční hodnoty (byl iniciován vždy se stejným semínkem) a byla tak zajištěna objektivita při třídění různými algoritmy v různých programovacích jazycích.

Při testování JNI rozhraní bylo v Javě alokováno pole, následně prostřednictvím JNI předáno ke zpracování programem zkompilovaným v dynamické knihovně, který pole naplnil, seřadil a seřazené pole vrátil zpět do javovské části programu. Čas byl měřen před alokací pole a po skončení nativní funkce, zároveň byl čas měřen v nativním kódu. Jelikož ale šlo o předávání referencí (v případě polí) nebo pouze jednociferných hodnot (n u výpočtu n-tého Fibonacciho čísla a počet u napojování řetězců), režie na předání dat v řádu milisekund nebyla měřitelná. Funkce `Fibonacci` a `StrCat` byla z javovského programu pouze zavolána s parametrem a zpracována programem uloženým v dynamické knihovně. Jednotlivé algoritmy byly spuštěny vždy desetkrát a výsledná hodnota času potřebného k jejich provedení byla vypočtena jako aritmetický průměr.

a) Quick sort – řadící algoritmus založený na třídění výměnou s rozdělováním. Problémem je optimální výběr pivota, který rozdělí tříděné prvky na dvě poloviny o stejném počtu prvků. Časová složitost je $O(n \log_2 n)$, avšak v nejhorším případě[9] až $O(n^2)$. [19]

b) Heap sort – třídění výběrem z haldy. Výhodou je, že pracuje na „in-situ“ a nepotřebuje žádný pomocný paměťový prostor ke zpracování dat. Má zaručenou časovou složitost $O(n \log_2 n)$. [19]

c) Insertion Sort – jedná se o řadící algoritmus založený na třídění přímým vkládáním. Časová složitost je $O(n^2)$. [19]

d) výpočet n-tého čísla Fibonacciho posloupnosti – Fibonacciho posloupnost je nekonečnou posloupností přirozených čísel, kdy kromě dvou prvních čísel je každé následující číslo součtem dvou předchozích. Algoritmus má lineární výpočetní složitost.

e) napojování řetězců – další možnost, jak ověřit rychlost provádění programu. V cyklu byly napojovány řetězce – vždy předchozí s novým (délky 7 znaků). V jazyce C++ byl použit datový typ `std::string` společně s funkcí `reserve` a operátorem +=. V Javě třída `StringBuffer` a funkce `append`.

Základy zdrojových kódu byly použity ze zdrojů [20], [21] a následně byly poupraveny.

[9] V případě, že pivot rozděluje posloupnost prvků na n-1 a 0 prvků.

6 Implementace vybrané dynamické knihovny do Java aplikace pomocí JNI

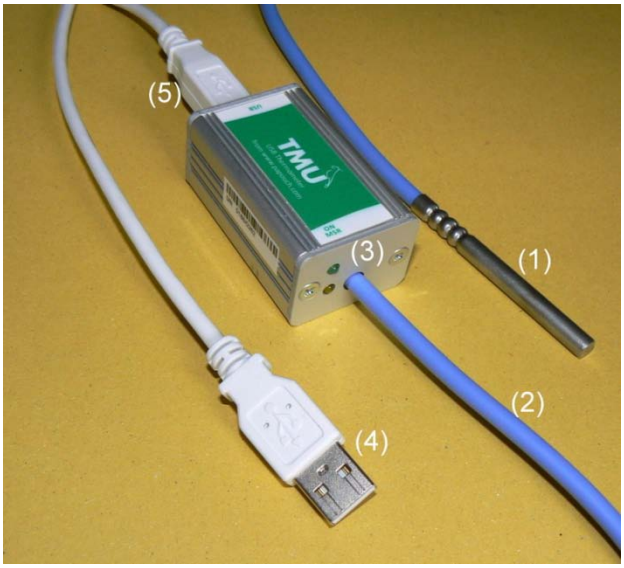
V teoretické části byly probrány základní možnosti využití JNI. V následujících kapitolách bude pojednáno, jak lze s pomocí JNI realizovat aplikaci, která ke své činnosti využívá nativní knihovnu dodanou od výrobce spolu s periferním zařízením.

6.1 Výběr aplikace

Zvolení konkrétní aplikace záleželo na omezeném výběru vhodných periferních zařízení. Jako ideální se naskytl teploměr s rozhraním USB, komunikující prostřednictvím protokolu Spinel [17].

6.1.1 Teploměr jako periferní zařízení

Na trhu je celá řada teplotních čidel, avšak jen málo z nich je schopných komunikovat s PC přes rozhraní USB a zároveň schopných poskytnout programátorovi další využití implementace. Jedním ze zařízení splňující tyto podmínky je USB teploměr TMU[10]. Je založen na polovodičovém teplotním čidle, schopném snímat teploty v rozsahu -55 °C až +125 °C. Samotné čidlo je propojeno s řídící elektronikou založenou na čipu firmy FTDI Chip[11]. Teploměr je dodáván od výrobce spolu s dynamickou knihovnou ovladače.



Obrázek 3 - Sestava teplotního čidla[12]

Na obrázku 3 je zachyceno samotné polovodičové teplotní čidlo (1), lisované v duralovém stonku normalizovaného průměru 6 mm a délky 60 mm. Teplotní čidlo je spojeno s krabičkou čipu silikonovým kabelem o průměru 4,3 mm s vysokou teplotní odolností (-60 až +200 °C) a délkou 3 m (2). Čip se nachází v krabičce z eloxovaného

[10] Více informací na http://www.papouch.com/cz/shop/product/tmu-usb-teplomer/
[11] Více informací na: http://www.ftdichip.com/
[12] Zdroj: vlastní zpracování

hliníku a na přední straně se nacházejí indikační LED diody (3). Na zadní straně je USB konektor typu B, sloužící k připojení propojovacího kabelu, vedoucího k počítači. Ten je na druhém konci zakončen USB konektorem typu A (4). [16, str. 5]

6.1.2 Komunikační protokol

Čip teploměru komunikuje s počítačem přes rozhraní USB protokolem Spinel [17, str. 20]. Tento je navržen pro využití v průmyslu, s cílem pokrýt kompatibilitu mezi jinými zařízeními. Je snadno implementovatelný ve všech běžných mikroprocesorech, programovatelných automatech a počítačích, umožňuje binární i ASCII komunikaci. Data v protokolu jsou přenášena pomocí rámců, kde je definován začátek a konec, viz tabulka 8. Informace o teplotě se nachází v bajtech číslo 1 – 5 (číslováno od 0), v bajtu číslo 6 je znaménko teploty.

Tabulka 8 – Formát protokolu [17]

Bajt	Znak	Význam
11	*	prefix
10	B	kód formátu
9	1	adresa teploměru
8	E	kód instrukce zařízení
7	1	
6	+	znaménko aktuální teploty
5	0	aktuální teplota
4	2	
3	1	
2	,	
1	5	
0	↵	zakončovací znak enter

Teploměr pravidelně odesílá prostřednictvím protokolu Spinel (typ 65) každých cca 10 vteřin datový rámec obsahující aktuální teplotu. [17, str. 20]

6.1.3 Komunikace s FTDI čipem

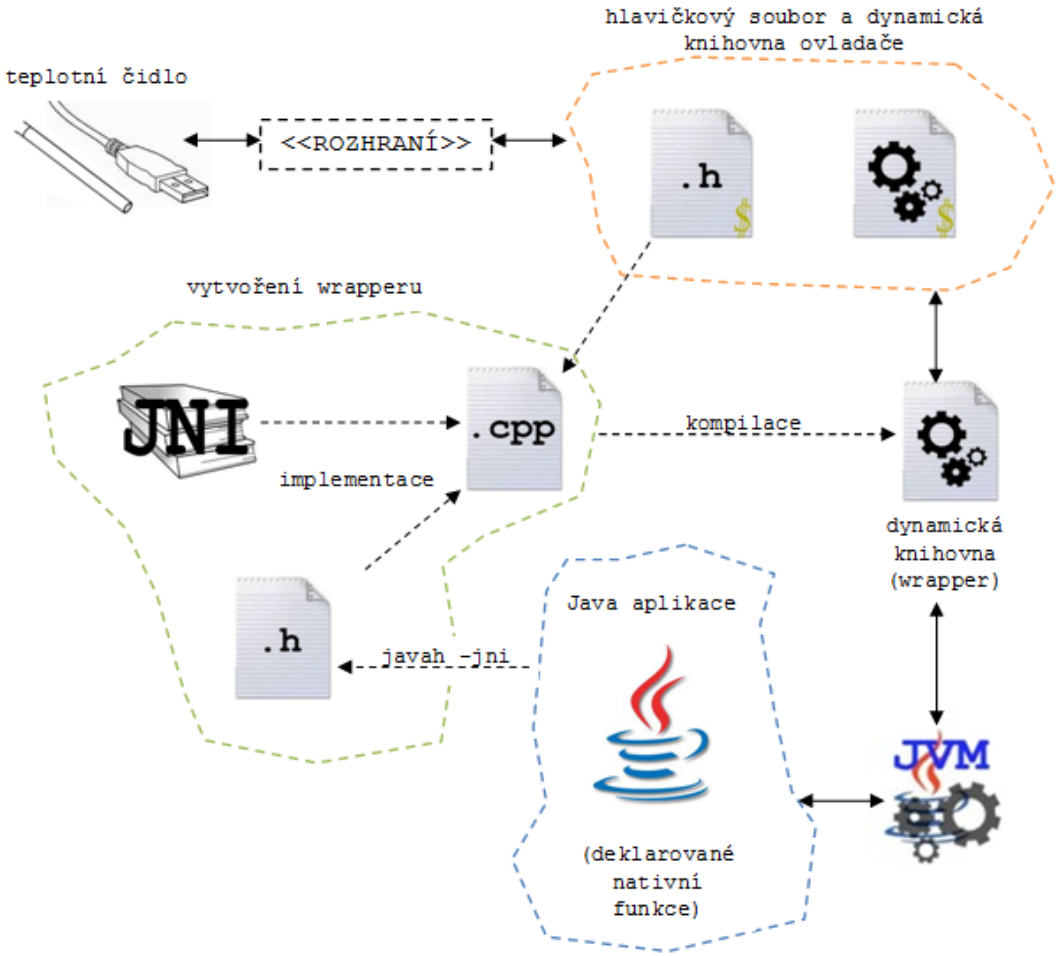
Součástí balení teploměru je i CD s nativní dynamickou knihovnou a hlavičkovým souborem, obsahujícím seznam deklarací funkcí, které lze z knihovny volat a ovládat tak řídící čip teplotního čidla. Ke knihovně existuje rozsáhlá dokumentace, obsahující seznam všech funkcí včetně popisu. K samotné komunikaci s čipem teplotního čidla je nezbytně nutné využít těchto funkcí [18]:

- `FT_STATUS` **`FT_Open`** `(int` *`iDevice`*`, FT_HANDLE *`*`ftHandle`*`)`
- `FT_STATUS` **`FT_Close`** `(FT_HANDLE` *`ftHandle`*`);`
- `FT_STATUS` **`FT_ListDevices`** `(PVOID` *`pvArg1`*`, PVOID` *`pvArg2`*`, DWORD` *`dwFlags`*`);`

- FT_STATUS **FT_Read** (FT_HANDLE *ftHandle*, LPVOID *lpBuffer*, DWORD *dwBytesToRead*, LPDWORD lpdwBytesReturned);
- FT_STATUS **FT_SetBaudRate** (FT_HANDLE *ftHandle*, DWORD *dwBaudRate*);
- FT_STATUS **FT_SetDataCharacteristics** (FT_HANDLE *ftHandle*, UCHAR *uWordLength*, UCHAR uStopBits, UCHAR uParity);
- FT_STATUS **FT_ResetDevice** (FT_HANDLE *ftHandle*);[13]

6.2 Návrh architektury aplikace

Aby bylo možné realizovat aplikaci pro obsluhu teploměru v programovacím jazyce Java, bude nutné využít JNI a jeho funkcí. Architekturu celého řešení z pohledu práce s jednotlivými knihovnami a částmi aplikace popisuje obrázek 4.



Obrázek 4 – Architektura řešení celé aplikace[14]

Z pohledu odspoda, tedy od periferního zařízení, bude s čipem teplotního čidla skrz rozhraní komunikovat aplikace díky dynamické knihovně ovladače, která implementuje

[13] Nejedná se o nezbytnou funkci pro komunikaci, ale v některých případech je vhodné zařízení resetovat. Seznam všech funkcí v [18].
[14] Zdroj: vlastní zpracování, MS Word

funkční rozhraní čipu dodané výrobcem. Aby bylo možné tyto funkce rozhraní volat, bude nutné implementovat nativní rozhraní mezi programovacím jazykem Java a funkcemi rozhraní čipu – tzv. wrapper. K jeho implementaci bude využita funkční sada čipu (deklarace funkcí z hlavičkového souboru), rozhraní JNI (předávání parametrů do prostředí JVM a zpět) a hlavičkový soubor vygenerovaný programem `javah` (obsahující deklarace nativních funkcí z Java části aplikace). Wrapper bude napsán v nativním jazyce C++ a následně z něj bude vytvořena dynamická knihovna, kterou načte část Java aplikace. Ta bude dále obsahovat GUI, ovládání teploměru, vykreslování grafu a export dat.

6.2.1 Zvolená vývojová prostředí

K vývoji celé aplikace bylo třeba implementovat kód v Javě i v nativním jazyce C++. Plnou podporu pro oba jazyky poskytuje vývojové prostředí NetBeans 7.0, konkrétně byla zvolena verze Beta (Build 201011152355).

Pro platformu Java nabízí tento program podporu pro práci s GUI, kde lze intuitivně navrhnout celý základní layout. Pro komplikovanější úpravy a nastavení layoutu je třeba implementovat kód ručně. Generovaný kód layoutu vytvořeného v designeru lze zobrazit, ale většinu nelze editovat.

Nativní část aplikace sice byla kompilována kompilátorem *cl*, jenž je součástí vývojového prostředí MS Visual Studio, ale z důvodu zachování pořádku ve složkách a projektech, byl kód psaný také v IDE NetBeans. Samotná kompilace pak proběhla v příkazovém řádku, který nabízí přímou kontrolu nad použitými parametry kompilace (např. použité optimalizace). Vývojové prostředí MS Visual Studio 2010 bylo ale použito při testování rychlostí prováděných algoritmů, popsaném v (4).

6.3 Návrh wrapperu

Wrapper, neboli obal, umožňuje díky JNI volat funkce nativní dynamické knihovny z prostředí JVM. Sám o sobě je dynamickou knihovnou načítanou z Javy.

6.3.1 Deklarace nativních metod

V Java třídě `JNIFTD2XX.java` jsou deklarovány pomocí klíčového slova `native` metody, které poskytují rozhraní Java části aplikace.

```
public native int read(byte[] bytes, int offset, int length)
throws Exception;
```

Pokud by došlo v nativní knihovně wrapperu k výskytu výjimky, bude zachycena a vrácena do Java aplikace jako instance třídy `Exception` (nebo její potomek). Tímto způsobem jsou ošetřeny všechny metody rozhraní. Z této třídy byl vygenerován programem `javah` hlavičkový soubor `jniftd2xx_JNIFTD2XX.h`, obsahující správné deklarace Java metod, tedy prototypy funkcí, které bude třeba implementovat v nativním jazyce C++.

6.3.2 Nativní část wrapperu

Implementace prototypů funkcí je v zásadě prosté zavolání funkcí rozhraní čipu teplotního čidla. Nicméně na této úrovni je také nutné správným způsobem předávat a vracet parametry funkcí z a do JVM, pomocí funkcí JNI správně zpracovat předávaná data, ošetřit potenciálně vzniklé výjimky a korektně vrátit jejich instance do JVM.

```
JNIEXPORT jint JNICALL
Java_jniftd2xx_JNIFTD2XX_read(JNIEnv *env, jobject obj, jbyteArray bPole,
jint offset, jint pocBajtu) {
    FT_STATUS status;
    volatile DWORD prectenoB;
    jint handle = getHandle(env, obj);
    int pocPrvkuPole = env->GetArrayLength(bPole);
    jbyte *buffer;

      if (bPole == 0) {
            jclass exc= env->FindClass("java/lang/NullPointerException");
            if (exc != 0) env->ThrowNew(exc, NULL);
            env->DeleteLocalRef(exc);
            return 0;
      } else if ((offset < 0) || (offset > pocPrvkuPole) || (pocBajtu <0)
            || ((offset + pocBajtu) > pocPrvkuPole)) {
            jclass exc =
                  env->FindClass("java/lang/IndexOutOfBoundsException");
            if (exc != 0) env->ThrowNew(exc, NULL);
            env->DeleteLocalRef(exc);
            return 0;
      } else if (pocBajtu == 0) return 0;

    buffer = env->GetByteArrayElements(bPole, 0);

    if (!FT_SUCCESS(status = FT_Read((FT_HANDLE) handle, (LPVOID) (buffer
+ offset), pocBajtu, (LPDWORD) &prectenoB))) {
        IOExceptionZprava(env, status);
    }
    env->ReleaseByteArrayElements(bPole, buffer, 0);
    return (jint) prectenoB;
}
```

Kód výše ukazuje implementaci jedné z nativních metod, konkrétně čtení daného počtu bajtů z čipu teplotního čidla. Prototyp funkce se skládá z maker `JNIEXPORT` a `JNICALL`, jejichž význam byl popsán v (2.4). Návratovým typem je `jint`, který nese informaci, kolik bylo přečteno z čipu bajtů. Následuje název funkce, obsahující název balíčku a Java třídy (`jniftd2xx.JNIFTD2XX`) a názvu metody (`read`). Parametry funkce jsou `JNIEnv` a `jobject` (2.4), dále reference na pole bajtů, kam budou uloženy přečtené bajty, offset a počet bajtů určených k přečtení (typicky velikost pole).

Pro komunikaci s čipem je nutné znát jeho aktuálně přidělený jednoznačný identifikátor, tzv. handle, který je získán po otevření spojení se zařízením. Následně je nutné někam ho uložit, protože jinak by nebylo možné zavolat žádnou z funkcí rozhraní čipu. Handle ale nelze v dynamické knihovně nikde uložit, protože se nejedná o třídu, ale pouze o funkce – je tedy nutné opět využít některé z funkcí JNI. To poskytuje díky funkcím `Set/Get<Typ>Field` přístup k instančním proměnným primitivních typů.

Hodnota handle je tedy uložena v datové struktuře Java třídy uvnitř JVM, reprezentovaná typem `jfieldID`. O operace s jednoznačným identifikátorem se starají v nativní implementaci wrapperu funkce `set/getHandle`.

Před samotným čtením bajtů z čipu je nutné provést základní ověření, zda jsou parametry funkce relevantní (kontrola velikosti pole, počtu bajtů určených k přečtení, atd.). Pokud daným parametrům nějaká hodnota nevyhoví, je nutné korektním způsobem informovat Java část aplikace – vrátit instanci správné třídy výjimky.

Pokud je funkce zavolána a pole `bPole` má nulovou délku, je tato situace ošetřena výjimkou a do Java aplikace navrácena instance třídy `NullPointerException`. Podobně, pokud jsou parametry funkce irelevantní, tzn., překračují povolené meze (záporný offset, záporný počet bajtů ke čtení apod.), je vrácena instance třídy `IndexOutOfBoundsException`. Po odeslání výjimky do JVM je nutné dealokovat třídu `exc`, která reprezentovala třídu výjimky.

Až po ověření všech potenciálně špatných situací je bezpečné přečíst z čipu daný počet bajtů. Funkce poskytované rozhraním čipu mají charakteristický prefix, a to `FT_`. Funkce `FT_Read` vrací informaci o tom, zda bylo čtení úspěšné, či nikoli. Při neúspěchu je opět nutné informovat část Java aplikace vzniklou výjimkou s odpovídající zprávou o dění v nativním kódu. Pro účely individuálních sdělení byla vytvořena funkce `IOExceptionZprava`, která má za úkol spárovat vrácenou chybovou hodnotu s odpovídající zprávou[15], zformátovat tuto zprávu a vrátit do Java části aplikace příslušnou instanci třídy (výjimky).

Pokud bylo čtení bajtů z čipu úspěšné, je již možné uvolnit alokované pole určené pro uložení přečtených bajtů – přečtené bajty jsou automaticky uloženy na adresu pole v JVM, jelikož pole předané funkci bylo předáno formou reference. Na konec funkce vrátí počet přečtených bajtů z čipu.

6.3.3 Kompilace do dynamické knihovny

Aby byla možná spolupráce Java části programu a nativní části programu, bylo nutné zkompilovat nativní funkce wrappeu do dynamické knihovny, kterou lze načíst Java částí aplikace. V rozsahu této práce byla provedena kompilace do dynamické knihovny typu dll kompilátorem *cl*, obdobným způsobem, jako bylo popsáno v (2.4.5).

[15] Funkce FT_Read (ale i jiné) vrací chybovou hodnotu v podobě čísla. V dokumentaci pak lze dohledat, o jaké chybě dané číslo informuje.

6.4 Spojení s nativní knihovnou z Java aplikace

Komunikace s funkcemi z nativní dynamické knihovny je zapouzdřena v Java třídě `JNIFTD2XX`. Po vytvoření instance této třídy je automaticky statickým iniciačním blokem načtena do paměti dynamická knihovna a vše je připraveno ke komunikaci.

6.4.1 Rozhraní mezi komunikační a aplikační částí

Třída `UsbInterface` implementuje veškerou logiku ovládání teploměru a prostřednictvím svého rozhraní nabízí funkce k jeho snadnému ovládání. Pomocí metody `jeZarizeniPripojeno` lze ověřit, zda uživatel připojil do USB teploměr. Tato metoda využívá nativní funkci `listDevices` ke zjištění počtu připojených zařízení. Protože je celá aplikace primárně určena pro provoz jednoho teplotního čidla, vrací tato funkce pouze stavy typu pravda nebo nepravda. Metoda `otevriZarizeni`, vracející informaci o úspěchu, nastaví všechny nutné parametry pro komunikaci s teploměrem – rychlost přenosu (9600 Baud), počet datových bitů (8), paritu (žádná) a počet stop bitů (1). Pokud žádná operace neskončila vzniklou výjimkou, bylo vše v pořádku nastaveno a zařízení lze považovat za úspěšně otevřené a připravené ke komunikaci. Podobně metoda `zavriZarizeni` informuje o úspěchu vráceným parametrem typu `boolean` a ve svém těle volá funkci `close` nativní knihovny (skrz třídu `JNIFTD2XX`).

Pro získání hodnoty teploty slouží metoda `getTeplota`. Po jejím zavolání proběhne nejprve ověření, zda je zařízení otevřené a připravené ke komunikaci (pomocí privátní proměnné třídy typu `boolean`) a zavolá se privátní metoda `ctiZarizeni`. Ta volá funkci `read` (skrz třídu `JNIFTD2XX`) nativní knihovny a do privátního pole typu byte o velikosti 12 prvků se uloží všechny přečtené bajty z čipu teplotního čidla. V metodě `getTeplota` je poté z informací obsažených v poli dekódována aktuální naměřená teplota (Tabulka 8 – Formát protokolu). Po úspěšném dekódování teploty je tato vrácena v datovém typu `double`. Pokud se vyskytne problém při konvergenci datového typu `String` na `double` (tzn. na pozicích bajtů udávající teplotu se objeví nečíselné znaky), je zachycena vzniklá výjimka (instance třídy `NumberFormatException`) a v ošetřujícím bloku je proveden reset čipu teplotního čidla.

6.4.2 Rutina obsluhující teplotní čidlo

Z teplotního čidla jsou periodicky odečítány hodnoty teploty a jsou ukládány v podobě nových instancí třídy `Bod` (6.5.2). Tato pravidelná rutina je prováděna ve vlákně. Po prvním spuštění vlákna je vytvořena nová instance třídy `UsbInterface` (spolu s ní je vytvořena i instance třídy `JNIFTD2XX`, viz výše) a v nekonečném cyklu je nejprve kontrolováno, zda je teplotní čidlo vůbec připojené k počítači metodou `jeZarizeniPripojeno` třídy `UsbInterface`. Tento stav je uložen ve výčtové třídě `enum` s názvem `StavyZarizeni`[16]. Vždy po kontrole je vlákno uspáno na jednu vteřinu.

[16] Možné stavy teplotního čidla jsou: připojeno, otevřeno a odpojeno.

Po úspěšné identifikaci teplotního čidla se přejde na další cykl, v němž se čeká, než uživatel stiskne tlačítko a započne měření teploty (signalizováno proměnnou `startMereni` typu `boolean`). Po tomto úkonu je zavolána metoda `otevriZarizeni` třídy `UsbInterface` (viz výše) a započne snímání teploty. Pokud by se nepodařilo v pořádku otevřít zařízení, je uživatel informován dialogovým oknem o neúspěchu a je vyzván, aby zkusil teplotní čidlo vypojit a znovu zapojit do USB.

Po úspěšném otevření zařízení následuje další cyklus, který běží po celou dobu měření a má za úkol kontrolovat, zda je spojení s teplotním čidlem otevřené a každých deset sekund z něj přečíst teplotu a uložit novou instanci třídy `Bod`. Poté je běh vlákna uspán na pět sekund. Čtení z čipu ale proběhne cca po deseti sekundách, protože čip má pevně nastavenou a neměnnou periodu měření. Tedy po přečtení bajtů se v jeho paměti žádná data nenacházejí a funkce `read` (6.3.2) skončí až po přečtení dat z paměti (čeká se na sejmutí nové teploty čipem). Pokud v tomto cyklu uživatel přeruší měření, je spojení s teplotním čidlem ukončeno a jeho stav je přepnut na `PRIPOJENO`.

Všechny tři výše popsané cykly jsou vnořeny do jednoho nekonečného cyklu. To znamená, že po přerušení měření uživatelem se znovu čeká, až uživatel opět zahájí měření a otevře se nové spojení s čidlem. Tato nekonečná smyčka může být přerušena pouze vzniklou výjimkou uvnitř některého z cyklů (při uspávání vlákna), nebo ukončením běhu vlákna, tedy ihned při ukončení programu (vlákno je typu démon).

6.5 Generování XML dat

Aby měla aplikace měřící teplotu nějaký smysl, je nutné data někam ukládat pro další zpracování. Vhodným řešením je formát XML, který splňuje požadavky nezávislosti na platformě, snadné přenositelnosti, univerzálnosti a má velkou podporu ze strany aplikací určených ke zpracování dat.

6.5.1 Sběr dat

Aplikace ukládá naměřená data do XML souboru formou průměrů za poslední počet měření, přičemž tento počet je nastavitelný uživatelem. Ten tak má možnost nastavit si ukládaný objem naměřených dat svým potřebám. Tato forma ukládání byla zvolena záměrně, aby ukládaný XML soubor nebyl příliš objemný a neobsahoval všechna měření s rozestupem cca 10 sekund.

Při probíhajícím měření se v obslužné rutině vlákna po úspěšném přečtení dat z teplotního čidla vždy uloží tato nově naměřená teplota. Ta je zapouzdřena spolu s dalšími daty ve třídě `Bod` a uložena do datové struktury seznamu. Současně s přidáním bodu metodou `pridejBod` je zavolána i statická metoda `kontrolaVelikostiDat` třídy `Body`, která zkontroluje, zda se v seznamu nachází daný počet[17] nezálohovaných instancí bodů – pokud ano, provede se záloha do XML souboru. Tato záloha je provedena i po ukončení

[17] Uživatelem definovaný v rozmezí 1 až 1000.

běhu aplikace. Každá instance třídy `Bod` si uchovává informaci, zda je již zálohovaná, či nikoliv.

6.5.2 Ukládání dat

Data ukládaná do souboru jsou vždy svázána s datem a časem. Od toho se odvíjí i zvolená struktura samotného XML dokumentu, jehož ukázka je v příloze. XML dokument, jakožto soubor, vzniká automaticky v závislosti na datu. Jeden den měření (nebo několik měření v jeden den) znamená jeden vytvořený soubor s XML dokumentem. Název souboru je ve formátu temp_RRRR-MM-DD. Algoritmus řešící tuto logiku je spolu s další veškerou funkcionalitou, spojenou s XML, implementován ve třídě `ZalohaDat` v balíčku `XML`.

Nové soubory XML jsou tvořeny pomocí třídy `SAXBuilder` z balíčku `jdom`.

```
doc = sb.build(cestaKSouboruXML);
```

Metoda `build` vytvoří nový prázdný XML soubor umístěný v cestě předané jako parametr, nebo pokud již existuje soubor na daném umístění, je nový dokument vystavěn na stávajícím, tzn. existující dokument je otevřen k editaci. Této skutečnosti je využito při zjišťování, zda je již dokument, vztahující se k dnešnímu datu vytvořený, nebo zda bude potřeba vytvořit nový soubor. Vyhodnocuje se název souboru, ze kterého je získáno datum a to je následně porovnáno s datem nejnovějšího bodu určeného k zapsání.

Naměřená teplota je uchovávána ve třídě `Bod` spolu s dalšími hodnotami. Konstruktorem třídy `Bod` je předána samotná teplota datovým typem `double` a v době vytvoření instance je zaznamenáno i časové razítko zapouzdřené do třídy `GregorianCalendar`. Dále třída `Bod` uchovává informace, zda je daný bod zálohovaný a zda je právě vykreslován v grafu. Oba tyto atributy třídy jsou typu `boolean`. Dále třída obsahuje metody pro vrácení formátovaného časového razítka typu `String` (formáty HH:MM a HH:MM:SS).

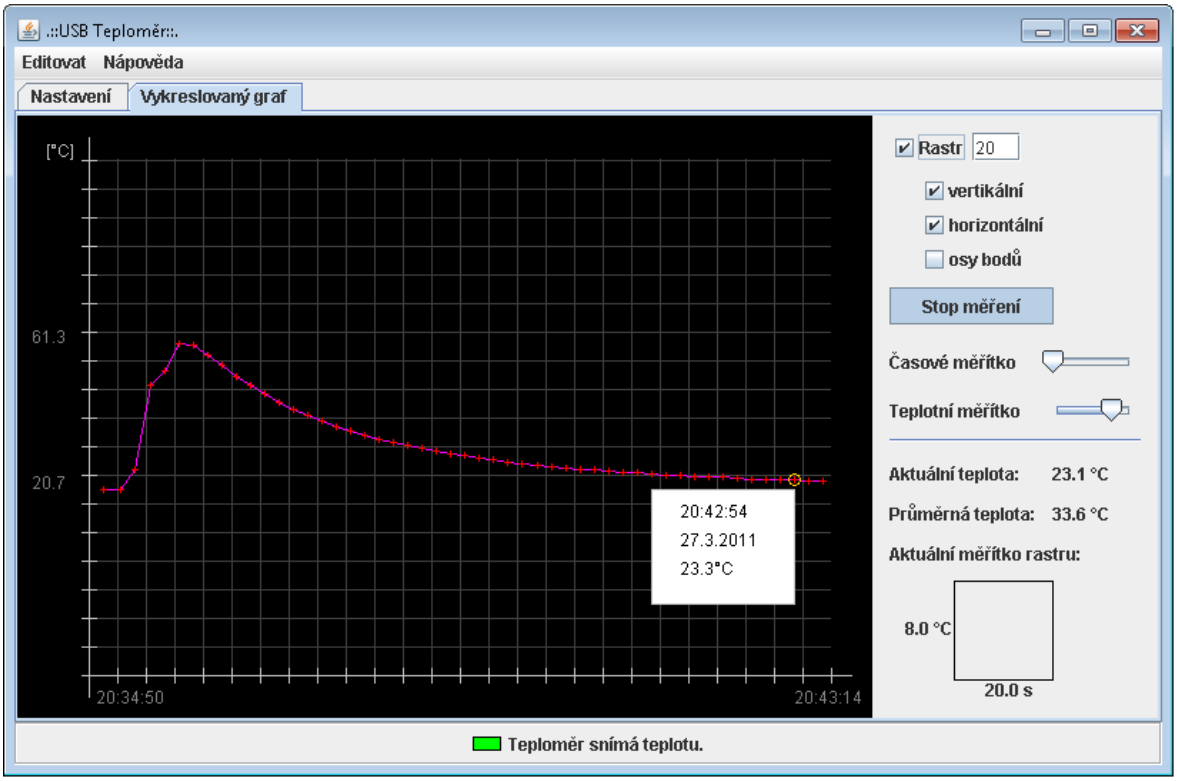
6.6 Grafické rozhraní aplikace

Základní vzhled grafického rozhraní aplikace byl navržen pomocí designeru vývojového prostředí NetBeans. Pokročilé úpravy layoutu byly doprogramovány.

6.6.1 Generování grafického výstupu

Požadavkem při návrhu aplikace byla možnost vykreslovat graf naměřených hodnot. Graf je koncipován tak, že s uložením nové instance třídy `Bod` do seznamu je plátno grafu překresleno a nová hodnota se objeví v pravé části grafického plátna. Jednotlivé body grafu jsou spojeny úsečkami. Uživatel má možnost upravovat měřítka časové i teplotní osy a graf je tak možné přizpůsobit měřeným teplotním rozsahům. Dále je uživatel informován o aktuální a průměrné teplotě. Informaci o konkrétním naměřeném bodě získá uživatel najetím myši do blízkosti požadovaného bodu, kde se poté zobrazí rámeček s naměřenými údaji – teplota, datum a čas pořízení teploty. Graf je možné doplnit o rastr zvolené velikosti. Ten dodá uživateli lepší orientaci v grafu a umožní mu vnímat lépe proporce grafu. S rastrem koresponduje i informace o aktuálním měřítku, vztahující se k aktuální velikosti jednoho dílku rastru, přepočítaná na skutečné jednotky. Dále je automaticky a dynamicky upravováno měřítko ve směru osy y v případě, že se objeví ve vykreslovaných bodech velký teplotní rozsah a nebylo by možné celý graf vykreslit. Vykreslovaná funkce je vystředěna na střed grafického plátna.

Uživatel má možnost v kartě nastavení měnit všechny barvy v grafu – body, spojnice, rastr, osy a pozadí. V této kartě se nachází i nastavení počtu měření, po kterých se uloží průměr za tato měření do XML souboru. Ve stavovém řádku dole je uživatel informován, v jakém stavu se aktuálně nachází teplotní čidlo vzhledem k programu. Existují tři možné stavy. Teplotní čidlo je odpojené od počítače (červená barva signalizace). Po připojení teplotního čidla k počítači se asi po 4 s rozsvítí oranžová signalizace, která znamená, že teplotní čidlo je správně připojené k počítači. Po stisknutí tlačítka start měření se po dalších cca 4 s rozsvítí zelená signalizace a je navázáno spojení s teplotním čidlem. Posléze se automaticky začne vykreslovat funkce naměřené teploty, viz obrázek 5.



Obrázek 5 - Vzhled grafického rozhraní[18]

6.6.2 Rozvržení panelů v okně

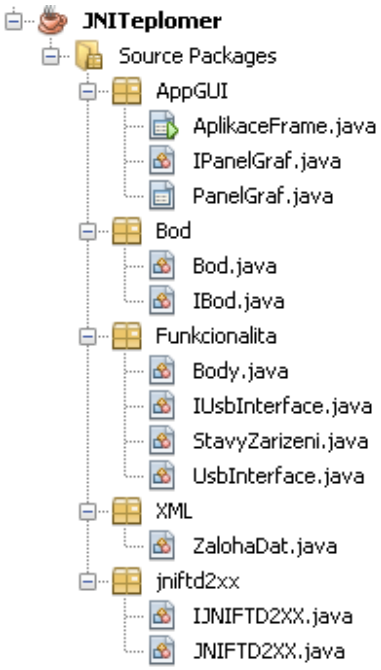
Komponenta grafu, nastavení na pravé straně i stavový řádek dole jsou instancemi třídy `JPanel`. Všechny jsou umístěny do rodičovské komponenty `JFrame` pomocí rozvržení `BorderLayout`. Komponenta grafického plátna a nastavení na pravé straně jsou spolu umístěny v jednom panelu, aby bylo možné nastavit nerovnoměrný poměr rozdělení, viz obrázek 5. K tomu byla použita třída `GridBagLayout` a pomocí statických atributů třídy `GridBagConstraints` nastaven nepravidelný layout celého kontejneru. Poměr stran grafického plátna je proměnný, ale šířka panelu nastavení zůstává stále pevně nastavena. Po změně velikosti grafického plátna je grafický obsah dynamicky aktualizován a překreslen, při zvětšení šířky se dokreslí dosud nezobrazené naměřené body (pokud jich je dostatek k dispozici).

[18] Zdroj: vlastní zpracování

6.7 Členění aplikace do tříd

Nativní část aplikace nebyla členěna ve vývojovém prostředí do projektu ani jiných speciálních adresářů, protože se jedná pouze o hlavičkový a zdrojový soubor. Zdrojové kódy byly navíc kompilovány do dynamické knihovny přímo z příkazové řádky a umisťování do projektu by dělalo situaci nepřehlednou.

Java část aplikace je situována v projektu vytvořeném v IDE NetBeans, logicky členěná do balíčků, které uchovávají třídy s logickou vazbou, viz obrázek 6.



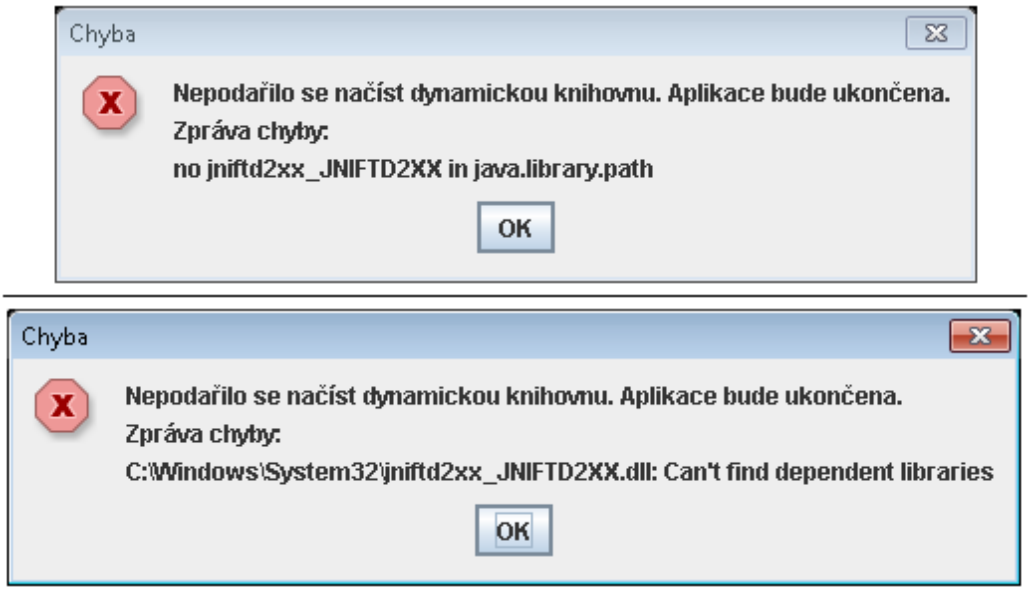
Obrázek 6 - Struktura Java aplikace z pohledu tříd[19]

V balíčku `AppGUI` jsou uchovány třídy, které zastávají funkci grafického výstupu. `XML` balíček uchovává třídy, pomocí kterých je prováděna záloha naměřených dat do XML souborů. V balíčku `Funkcionalita` jsou uloženy hlavní třídy, pomocí kterých je prováděna komunikace mezi třídami s GUI a rozhraním pro komunikaci s nativní knihovnou (`UsbInterface` spolu se svým rozhraním), dále výčtová třída `enum` (`StavyZarizeni`) a třída Body, která uchovává veškeré statické metody pro práci s naměřenými daty a metody, které slouží k pomocným výpočtům pro grafický výstup. V balíčku `jniftd2xx` se nachází třída s nativními metodami, které jsou implementovány v nativním jazyce (6.3).

[19] Zdroj: vlastní zpracování

6.8 Spuštění aplikace

Aplikaci lze spustit na počítači, který má nainstalované JRE. Nicméně aby bylo možné korektně navázat spojení s teplotním čidlem, je nutné do vyhledávacích cest[20] umístit dynamickou knihovnu wrapperu a dynamickou knihovnu ovladače teplotního čidla. Pokud do těchto cest nejsou knihovny jniftd2xx_JNIFTD2XX.dll a ftdi2xx.dll nakopírovány nebo nejsou cesty správně nastaveny, bude při spuštění aplikace zachycena výjimka a uživatel bude informován o chybě. Obrázek 7, nahoře – program nemůže načíst dynamickou knihovnu jniftd2xx_JNIFTD2XX.dll, obrázek 7, dole – programu se nepodařilo načíst dynamickou knihovnu ovladače ftd2xx.dll.



Obrázek 7 - Možná chybová hlášení při spuštění aplikace[21]

Aplikace byla testována v prostředí OS Windows a pro použití např. v OS Linux by bylo ještě třeba překompilovat nativní wrapper pod platformou Linux a dále použít správnou dynamickou knihovnu ovladače teplotního čidla, určenou pro danou platformu operačního systému.

[20] Vyhledávací cesty systému spravuje proměnná PATH.
[21] Zdroj: vlastní zpracování

7 Závěr

Provedenými pokusy jsem zjistil, že pomocí JNI lze náročné části Java aplikace efektivně implementovat v nativním jazyce. Tento nativní kód lze poté dobře optimalizovat a dosáhnout tak lepších výpočetních výsledků implementovaných algoritmů. Režie na volání nativních funkcí z prostředí programovacího jazyka Java je minimální. Pokud jsou data předávána pomocí referencí, je čas potřebný k předání dat zanedbatelný a nativní kód tak pracuje přímo s nimi, nevytváří se žádné kopie.

Po volbě vhodného kompilátoru (*cl*) s optimalizacemi, byly všechny testované třídící algoritmy v nativním jazyce C++ provedeny téměř dvakrát rychleji, než tomu bylo v programovacím jazyce Java. Pouze rekurzivně implementovaný algoritmus na výpočet n-tého čísla Fibonacciho posloupnosti byl proveden v Javě rychleji. Jen kompilátor *gcc* na platformě Linux byl rychlejší i přes to, že jeho varianta na platformě Windows měla stejné hodnoty nastavení při kompilaci do dynamické knihovny. Z toho plyne, že při vývoji robustní aplikaci v Javě, která implementuje výpočetně náročné algoritmy, se vyplatí přemýšlet o možnosti využít výhod JNI a optimalizovaného nativního kódu.

Při vývoji aplikace, využívající ke své funkcionalitě nativní knihovnu, poskytuje JNI programátorovi rozhraní mezi javovskou a nativní částí programu. Po pochopení filozofie fungování JNI a nastudování jeho funkcí a typů, se mi s rozhraním JNI pracovalo intuitivně. Malou překážkou pro mě byla pouze implementace wrapperu, kde jsem musel pracovat přímo s funkcemi čipu a programoval na nízké úrovni. Bylo obtížnější hledání chyb a ladění nativní části, protože wrapper je kompilován do dynamické knihovny a tím je komplikována možnost krokování programu.

Aplikaci JNI, jako prostředku ke komunikaci s periferiemi, jsem využil z důvodu spojit nativní knihovnu ovladače s programovacím jazykem Java, tedy výhody rychlejšího nativního kódu byly až na druhém místě. Ze získané zkušenosti bylo vhodné v nativním kódu implementovat jen nezbytně nutné minimum a převážnou část funkcionality tak přesunout až na vyšší úroveň, tedy do Javy. Měl jsem tak nad kódem mnohem větší kontrolu a mohl ho lépe ladit.

Literatura

[1] NOVOTNÝ, Luděk . *Sylaby k předmětu* [online]. 2003 [cit. 2011-04-04]. Historie a vývoj jazyka Java. Dostupné z WWW: <http://www.fi.muni.cz/usr/jkucera/pv109/2003p/xnovotn8.htm>.

[2] DOLEŽEL, Luboš. *AbcLinuxu* [online]. 21. 10. 2010 [cit. 2011-04-04]. Java Native Interface: propojujeme Javu a C/C++ – 1. Dostupné z WWW: <http://www.abclinuxu.cz/clanky/java-native-interface-propojujeme-javu-a-c-cplusplus-1>.

[3] LIANG, Sheng. *The Java™ Native Interface : Programmer's Guide and Specification*. Reading (Massachusetts) : Addison-Wesley, 1999. Dostupné z WWW: <http://java.sun.com/docs/books/jni/download/jni.pdf>. ISBN 0-201-32577-2.

[4] Java.net [online]. c2011 [cit. 2011-04-04]. Java Native Access (JNA): Pure Java Access to Native Libraries. Dostupné z WWW: <http://jna.java.net>.

[5] DOLEŽEL, Luboš. AbcLinuxu [online]. 10. 11. 2010 [cit. 2011-04-04]. Java Native Interface: propojujeme Javu a C/C++ – 2. Dostupné z WWW: <http://www.abclinuxu.cz/clanky/java-native-interface-propojujeme-javu-a-c-cplusplus-2>.

[6] IBM. *Publib.boulder.ibm.com* [online]. c2011 [cit. 2011-04-28]. Using input and output streams for interprocess communication. Dostupné z WWW: <http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=%2Frzaha%2Fiostream.htm>.

[7] Oracle. *Oracle Technology Network* [online]. c2011 [cit. 2011-04-28]. Socket Communications. Dostupné z WWW: <http://www.oracle.com/technetwork/java/socket-140484.html>.

[8] *Java.net* [online]. c2011 [cit. 2011-04-04]. Java Native Access (JNA): Pure Java Access to Native Libraries. Dostupné z WWW: <http://jna.java.net>.

[9] JONES, M. Tim. *IBM Developer Works* [online]. 2008-08-20 [cit. 2011-04-10]. Anatomy of Linux dynamic libraries. Dostupné z WWW: <http://www.ibm.com/developerworks/linux/library/l-dynamic-libraries/>.

[10] KOSEK, Jiří. *XML pro každého: podrobný průvodce*. 1. vydání. Praha : Grada, 2000. 164 s. Dostupné z WWW: <http://www.kosek.cz/xml/xmlprokazdeho.pdf>. ISBN 80-7169-860-1.

[11] HÉGARET, Philippe Le. *Document Object Model (DOM)* [online]. 2005-01-19, 2009-01-06 [cit. 2011-04-07]. Document Object Model (DOM). Dostupné z WWW: <http://www.w3.org/DOM/>.

[12] MEGGINSON, David. *Megginson Technologies* [online]. c1998–2011 [cit. 2011-04-08]. Simple API for XML. Dostupné z WWW: <http://www.megginson.com/downloads/SAX/>.

[13] FRY, Chris; SLOMINSKI, Aleksander. *The Codehaus* [online]. c2003-2006 [cit. 2011-04-08]. The Streaming API for XML (StAX). Dostupné z WWW: <http://stax.codehaus.org/Home>.

[14] HUNTER, Jason. *JDOM* [online]. c200-2007 [cit. 2011-04-08]. Mission. Dostupné z WWW: <http://www.jdom.org/mission/index.html>.

[15] PATRNÝ, Vojtěch. *Linuxzone* [online]. 2002-09-17 [cit. 2011-04-08]. Programově na XML, 3.díl - JDOM. Dostupné z WWW: <http://www.linuxzone.cz/index.phtml?ids=2&idc=377>.

[16] Papouch s.r.o. *USB teploměr TMU*. Praha: Papouch s.r.o., 2005. 20 s. Dostupné z WWW: <http://www.papouch.com/cz/shop/product/tmu-usb-teplomer/tmu.pdf/_downloadFile.php>.

[17] Papouch s.r.o. *Papouch* [online]. c2006-2011 [cit. 2011-04-24]. Spinel. Dostupné z WWW: <http://www.papouch.com/cz/website/mainmenu/spinel/>.

[18] Future Technology Devices International Ltd. *Software Application Development: D2XX Programmer* [online]. UK : [s.n.], 2010-11-04 [cit. 2011-04-17]. Dostupné z WWW: <http://www.ftdichip.com/Support/Documents/ProgramGuides/D2XX_Programmer>.

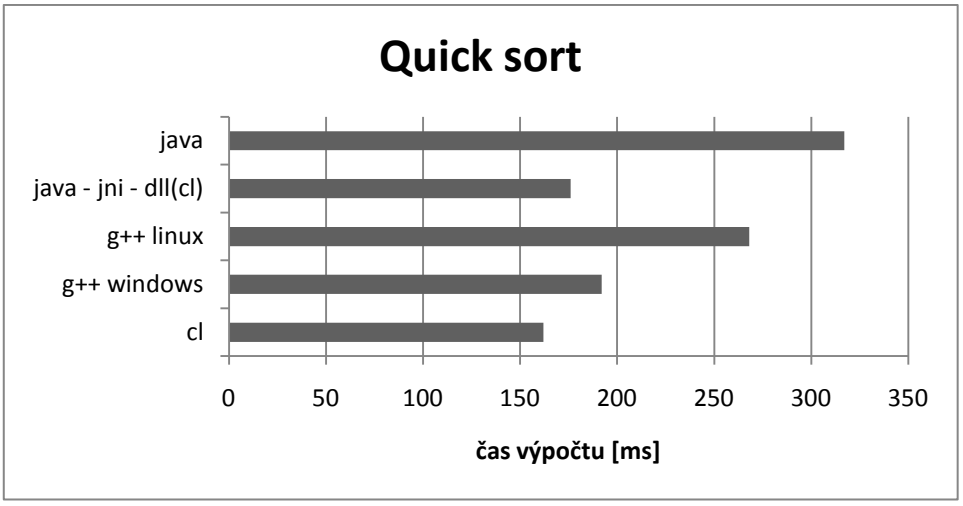
[19] KAVIČKA, Antonín. *Sylaby k předmětu IDATS*. Pardubice : [s.n.], 2010. Algoritmy třídění, s. 2-11.

[20] NOVÁK, Pavel. *Pavel Novák blog: Algoritmy* [online]. 2008 [cit. 2011-05-02]. Dostupné z WWW: <http://pavel-novak.net/algoritmy/>.

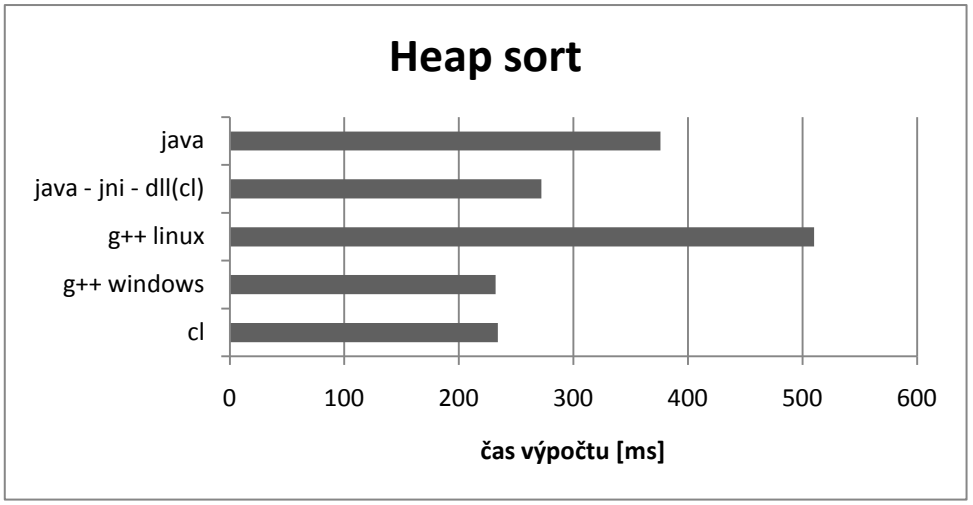
[21] LEA, Keith. *The Java is Faster than C++ and C++ Sucks Unbiased Benchmark* [online]. 2003-11-14 [cit. 2011-05-02]. Dostupné z WWW: <http://keithlea.com/javabench/>.

[22] SIERRA, Kathy; BATES, Bert. *Head First: Java*. 2nd edition. [s.l.] : O, 2005. 688 s. ISBN 978-0-596-00920-5.

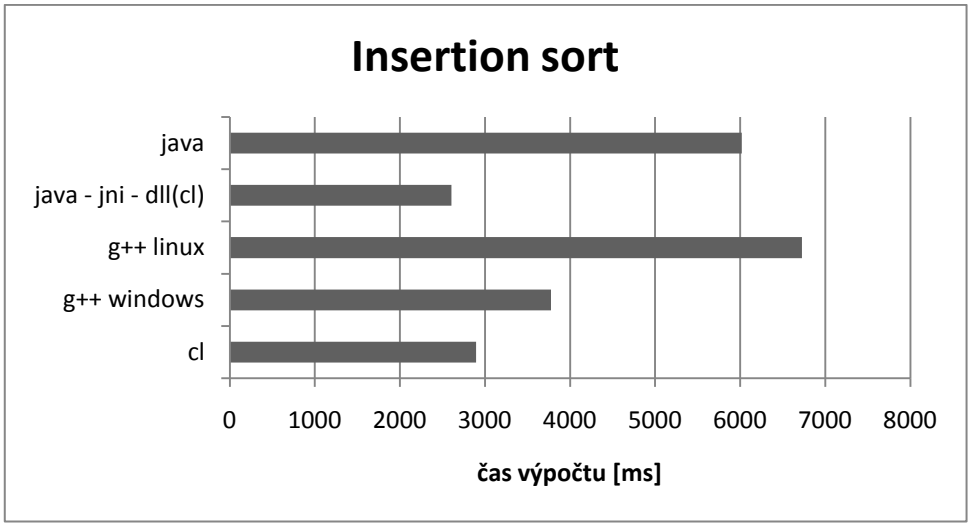
Příloha A – Grafy popisující měření rychlostí algoritmů



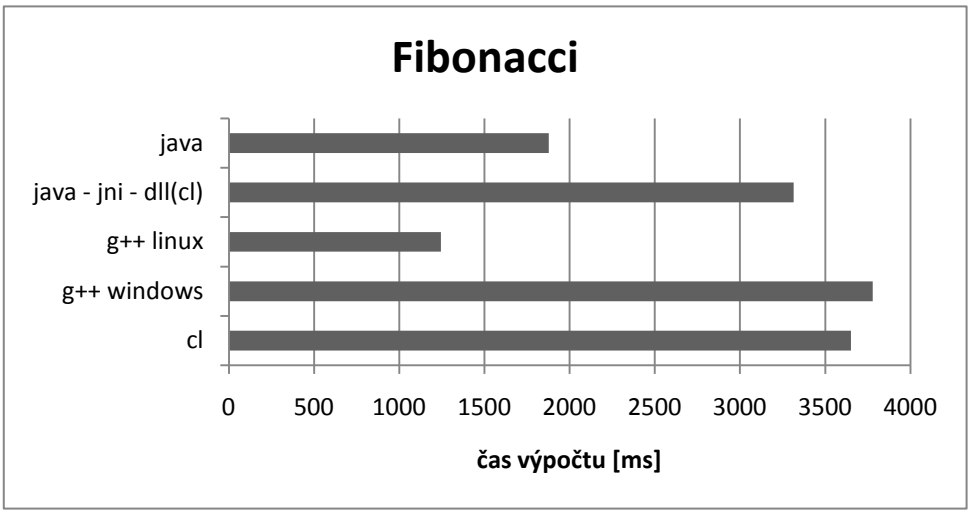
Graf 2 - Časy seřazení algoritmem Quick sort



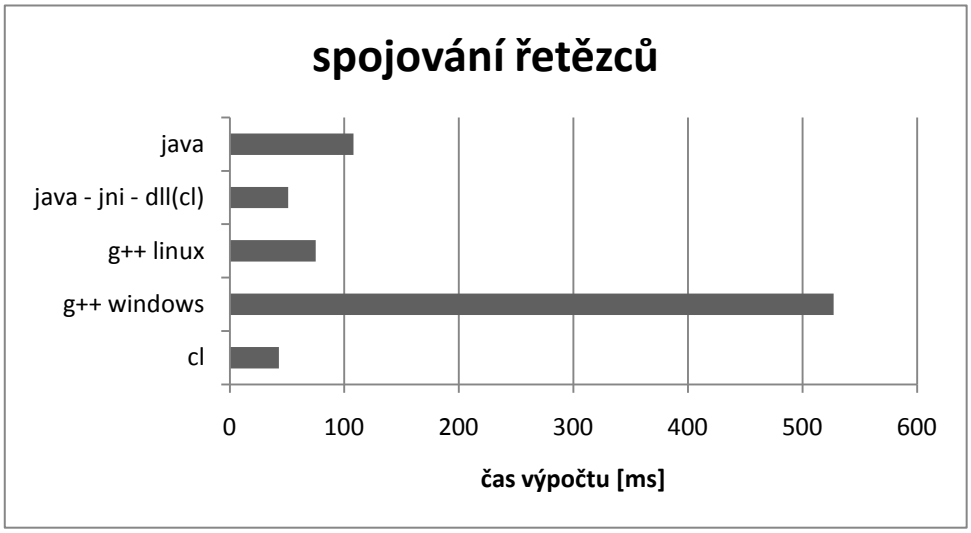
Graf 3 - Časy seřazení algoritmem Heap sort



Graf 4 - Časy seřazení algoritmem Insertion sort



Graf 5 - Časy výpočtu 40tého čísla Fibonacciho posloupnosti



Graf 6 - Časy potřebné k napojení 10^6 řetězců

Příloha B – Ukázka formátu výstupního XML souboru

```
<?xml version="1.0" encoding="UTF-8"?>
<temperature_meter>
  <info>JNI USB teploměr</info>
  <units>°C</units>
  <date idd="temp_2011-4-30.xml">
    <time idt="13:32:02">
      <temperature>36.79</temperature>
    </time>
    <time idt="13:34:43">
      <temperature>28.96</temperature>
    </time>
    <time idt="13:35:13">
      <temperature>25.5</temperature>
    </time>
    <time idt="13:35:33">
      <temperature>24.95</temperature>
    </time>
    <time idt="13:36:23">
      <temperature>24.4</temperature>
    </time>
    <time idt="13:37:14">
      <temperature>23.74</temperature>
    </time>
    <time idt="13:38:04">
      <temperature>23.34</temperature>
    </time>
    <time idt="13:38:04">
      <temperature>29.25</temperature>
    </time>
    <time idt="13:38:14">
      <temperature>23.1</temperature>
    </time>
    <time idt="13:38:24">
      <temperature>37.1</temperature>
    </time>
    <time idt="13:38:34">
      <temperature>44.7</temperature>
    </time>
    <time idt="13:38:44">
      <temperature>45.1</temperature>
    </time>
    <time idt="13:38:54">
      <temperature>44.4</temperature>
    </time>
    <time idt="13:39:04">
      <temperature>43.3</temperature>
    </time>
    <time idt="13:39:14">
      <temperature>42.2</temperature>
    </time>
    <time idt="13:39:24">
      <temperature>40.7</temperature>
    </time>
  </date>
</temperature_meter>
```