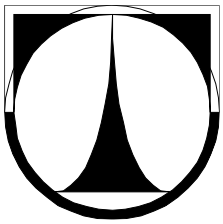
TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií



DIPLOMOVÁ PRÁCE

Liberec 2011 **Martin Blížkovský**

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 3902T005– Automatické řízení a inženýrská informatika

Optimalizace systémové služby pro automatický záznam dat

Optimization of system service for automatic data recording

Diplomová práce

Autor: **Bc. Martin Blížkovský**
Vedoucí práce: Ing. Jan Kraus
Konzultant: Ing. Tomáš Tobiška

V Liberci 12.5.2011

ZDE BUDE VLOŽENO ORIGINÁLNÍ ZADÁNÍ PRÁCE

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé DP a prohlašuji, že **s o u h l a s í m** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Jsem si vědom toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum

Podpis

Abstrakt

Cílem tohoto projektu je optimalizovat kód systémové služby, která pracuje ve vícevláknovém provozu. Její základní podobu jsem navrhl ve svých předchozích projektech, kterými byla má bakalářská práce [3] a semestrální projekt [4]. Služba slouží k automatickému cyklickému stahování dat z měřících přístrojů společnosti KMB systems, s.r.o. a jejich následnému ukládání do databáze. Za účelem optimalizace jsem musel rozšířit své znalosti v oblasti tvorby paralelních aplikací v prostředí .NET. Jako například seznámit se s různými konstrukčními prvky, které slouží k jejich tvorbě. Dále bylo také nutné seznámit se s dostupnými metodami, které umožňují analýzu a měření výkonnosti takových aplikací a napomohou tak k odhalení kritických míst v současném řešení.

Po seznámení se s metodami tvorby paralelních aplikací a výběru vhodných metod, které slouží k měření výkonnosti, byla systémová služba podrobena testům za účelem její analýzy. Testy byly provedeny při různých úrovních vytížení, kterými bylo množství použitých měřících přístrojů a množství cyklických operací, zajišťujících záznam dat. Po analýze samotné systémové služby byly dále analyzovány i dostupné metody, zajišťující paralelizaci aplikace, za účelem výběru nejvhodnější metody vzhledem k potřebám systémové služby. Na základě vyhodnocení výsledků jednotlivých testů byly realizovány rozsáhlé úpravy současné struktury aplikace za účelem navýšení výkonnosti. Systémová služba s novými úpravami byla opět analyzována a některé výsledky porovnány se starší neoptimalizovanou verzí. Nakonec byla do systémové služby doplněna základní podpora rozhraní pro vzdálený monitoring v podání SNMP.

Vytvořená aplikace disponuje v porovnání se starou verzí vylepšenou správou vláken, ve kterých dochází k vykonávání zadaných operací. To má za následek několikanásobně nižší nároky na operační paměť. Dále bylo také vlivem provedených úprav sníženo vytížení procesoru. Systémová služba je také schopna poskytovat informace o své činnosti prostřednictvím zpráv protokolu SNMP.

Klíčová slova: Systémová služba, Paralelní programování, Optimalizace aplikace, Profilování aplikace

Abstract

Main goal of this project is optimization of system service code, which works with multiple threads. Its basic form was designed in my previous projects, such was my bachelor thesis [3] and term project [4]. Service is used for automatic cyclic data recording from KMB systems, s.r.o measuring instruments and then data are stored to database. In order of optimization I had to extend my knowledge in the area of designing parallel applications in the .NET environment. Like for example become familiar with various elements that are used to create them. It was also necessary to get in touch with methods that allows analysis and measuring of application performance and thus helps to reveal bottlenecks of current solution.

After familiarization with methods for creating parallel applications and selecting appropriate methods for measuring application performance, system service was tested for purpose of further analysis. Tests were conducted under different workloads which means number of involved instruments and number of cyclic operations which secures recording of data. After analysis of system service, other tests were performed on methods that are in touch with parallelization, in order to select best method for needs of system service. On the basis of the evaluation of the tests results, present structure of application were undertaken large revision to increase its performance. System service with new modifications was again analyzed and some of the results were compared with older non-optimized version. Lastly, the system service was equipped with basic support of interface for remote monitoring by SNMP.

Final application has, in comparison to the older version, improved thread management, which are assigned to carry out requested operations. This has resulted in much lower demands on main memory. Reduced usage of CPU was another result of adjustments. System service is now also able to provide information about its activities through reports via SNMP.

Keywords: System service, Parallel programming, Optimization of application, application profiling

Obsah

1 Úvod....9
2 Paralelní programování....11
2.1 .NET....11
2.2 .NET 4....12
2.3 Konkurence....14
3 Testování výkonnosti, profilování....16
3.1 Výkon (Performance)....16
3.1.1 Profillery....16
3.1.2 Výkonnostní čítače....16
3.2 Spolehlivost (Reliability)....18
3.3 Škálovatelnost (Scalability)....18
4 Popis aplikace....20
4.1 Podoba současné aplikace....20
5 Analýza a měření parametrů....23
5.1 Parametry k měření....23
5.2 Problémy odhalené při analýze....23
5.3 Návrh optimalizované podoby aplikace....24
5.3.1 Řešení 1 – Centrální správce operací....25
5.3.2 Řešení 2 – Decentralizovaný systém s časovači....27
6 Technická řešení nové struktury....30
6.1 Analýza možných metod synchronizace sdílených dat....30
6.2 Výsledky testů synchronizace sdílených dat....33
6.3 Analýza dostupných metod umožňujících paralelizaci aplikace....37
6.4 Výsledky testu paralelizačních metod....39
7 Dosažené výsledky....47
7.1 Využití paměti....47
7.2 Využití procesoru....50
7.3 Doba potřebná pro načtení parametrů....51
7.4 Shrnutí....53
7.5 Možnosti budoucího rozvoje....53
8 SNMP....55

8.1 Popis protokolu 55
8.1.1 Formát zprávy a její předávání.....55
8.2 Implementace SNMP.....57
9 Závěr.....59
Seznam použité literatury.....61
Příloha A – Sledování výkonnostních čítačů.....63
Příloha B – Klientská aplikace.....65
Příloha C – Instalace aplikace.....67

Seznam zkratek a termínů

OS – Operační systém

VS – Visual Studio

SNMP – Simple Network Management Protocol

PDU – Protocol Data Unit / Datová Jednotka Protokolu

OID – Object Identifier / Identifikátor objektu

CLR – Common Language Rutine

CIL - Common Intermediate Language

ASN.1 – Abstract Syntax Notation One

TPL – Task Parallel Library

LINQ – Language Integrated Object

GUI – Graphic User Interface / Grafické uživatelské rozhraní

IPC – Inter-Process Communication / Mezi-procesová komunikace

ORM - Object-relational mapping/Objektově relační mapování

MIB – Management Information Base

Profiller – nástroj monitorující využití zdrojů aplikace za jejího chodu

1 Úvod

Podstatou mé diplomové práce je nastudovat dostupné metody tvorby vícevláknových aplikací v rozhraních .NET operačního systému Windows. Následně s použitím získaných znalostí o tvorbě vícevláknových aplikací a na základě analýzy současné struktury aplikace optimalizovat kód systémové služby pro automatický záznam dat. Účelem takové optimalizace je snížení náročnosti optimalizované aplikace na systém a tím navýšení její výkonnosti. To umožní aplikaci zpracovávat současně více požadavků při zachování obdobných nároků na výkon systému. Mimo to bylo nutné seznámit se s dostupnými možnostmi implementace rozhraní pro vzdálený monitoring a správu.

Optimalizace spočívá v úpravě podoby současné struktury systémové služby s použitím nových nebo vhodnějších metod, umožňujících paralelizaci aplikace. Úpravy jsou prováděny na základě analýzy současné aplikace použitím vhodných způsobů, které měří využití systémových zdrojů a celkové chování aplikace. Zmiňovaná analyzovaná systémová služba je vícevláknová vzdáleně řízená aplikace, jejíž základní strukturu jsem navrhl v rámci bakalářské práce. Hlavní činností služby je cyklické stahování dat z měřicích přístrojů a následné ukládání do databáze.

Pro analýzu se používají nástroje zvané profillery, které mají za úkol sledovat aplikaci za chodu a zaznamenávat potřebné parametry. K analýze je také možné provázat použití profilleru s výkonnostními čítači. Čítače nabízejí sledování jak systémových dat souvisejících s analyzovanou aplikací, tak i vytvoření vlastních, které je možné integrovat přímo na požadovaná místa v aplikaci a měřit tak parametry jako rychlost a podobně. Vyjma analýzy samotné aplikace je třeba provést také analýzu metod využívaných při paralelizaci aplikace. Každá z těchto metod je vhodná na jiná použití a disponuje jinými vlastnostmi.

Celá aplikace systémové služby je vytvořena s použitím programovacího jazyka C# prostřednictvím vývojového a programovacího prostředí Visual Studio od společnosti Microsoft. Před optimalizací využívala aplikace knihovny z rozhraní .NET 2.0 a byla vytvořena konkrétně v programovacím prostředí VS 2008. Optimalizovaná verze využívá některé nové metody z knihoven .NET 4.0 a s tím je spojen i přechod na novou verzi Visual Studio 2010. Úpravy ve struktuře provedené na základě analýz

jednotlivých komponent spočívají v náhradě určitých používaných metod novými vhodnějšími metodami a také v realizaci nových systémů a struktur některých kritických částí aplikace. Provedené změny mají za úkol navýšit celkovou výkonnost aplikace a snížit její požadavky na systém. Nakonec bylo také nutné do systémové služby implementovat podporu protokolu SNMP umožňující vzdálený monitoring chodu aplikace.

Výsledné a původní řešení systémové služby je nakonec vhodné podrobit sérii různých testů za účelem ověření správnosti nového návrhu. Takovéto testy by měly prokázat snížení náročnosti na systém, případně nárůst výkonnosti nové podoby systémové služby.

2 Paralelní programování

Význam paralelního programování aplikací v současnosti značně narůstá z důvodů velkého rozšíření hardwaru schopného vykonávat paralelní úlohy. Dříve se výpočetní výkonnost zvyšovala téměř výhradně pouze zvyšováním frekvence CPU. Tento trend se však již před několika lety zastavil a v dohledné době se nepředpokládá, že by frekvence měla nějak výrazněji narůstat. Výkonnost procesoru se tedy nyní navyšuje počtem jader, která procesor obsahuje. Avšak nárůst výkonu je znát až s paralelními aplikacemi, které jsou schopné dostatečně využít schopnosti dostupného paralelního hardwaru.

Výkonnost paralelní aplikace by měla narůstat s možnostmi paralelního hardwaru (vícejádrové CPU), ale současně by měla být její výkonnost ekvivalentní se sekvenční aplikací v případech, kdy není paralelní hardware k dispozici, jako jsou počítače s jedním jádrem procesoru.

2.1 .NET

.NET je softwarová platforma pracující primárně pod OS Windows. Poprvé byla dostupná ve verzi Windows NT. Podporuje několik programovacích jazyků a také jejich vzájemnou zaměnitelnost, kdy každý jazyk podporující .NET může používat i části kódu z jiného jazyka podporujícího .NET. Programy vytvořené s použitím těchto jazyků jsou přeloženy do jakéhosi univerzálního mezijazyka CIL. .NET dále disponuje velkým balíkem knihoven, které jsou dostupné pro všechny podporované jazyky.

Podpora tvorby paralelních aplikací byla v .NET od samého počátku. Od počátku bylo nabízeno také velké množství metod za tímto účelem vytvořených, které existují v nových verzích .NET dodnes. Metody umožňující paralelizaci aplikace jsou například Thread, ThreadPool se svojí frontou pro asynchronní úkoly nebo asynchronní použití delegáta. Disponoval také většinou současných metod pro synchronizaci přístupu ke společným datům. Schopnosti některých metod se však s postupným vývojem nových verzí .NET měnily. Například ThreadPool uživateli v prvních verzích nabízel k dispozici v základu 25 vláken na jádro procesoru. Hodnotu sice bylo možné změnit, ale ne žádným komfortním způsobem. V poslední verzi .NET tato metoda nabízí vlákna v závislosti na konfiguraci systému a ve většině případů přesahuje jejich

počet 1000 vláken na jádro. S příchodem .NET verze 2.0 byly doplněny některé nové metody jako například Background worker. Významnější změny v přístupu ke tvorbě paralelních aplikací v .NET přinesla zatím poslední verze 4. Zmíněnými změnami jsou hlavně dva nové moduly Task Parallel Library a PLINQ.

2.2 .NET 4

Z těchto důvodů, v době kdy vývoj HW klade čím dál tím větší nároky na korektně vytvořené paralelní aplikace, vydala společnost Microsoft platformu .NET 4 spolu s vývojovým prostředím Visual Studio 2010. To rozšiřuje tvorbu paralelních aplikací o nové metody, knihovny, rozšířené možnosti ladění paralelních aplikací a další.

Mezi nejdůležitější novinky umožňující kvalitnější tvorbu paralelních aplikací patří technologie Task Parallel Library (TPL). Jedná se o jakousi nástavbu nad multi-threadovým modelem, který byl dostupný již v předchozích verzích .NET. Hlavním prvkem TPL je task (úkol). Nepřímo srovnatelným ekvivalentem byly v dřívějších verzích .NET vlákna (threads). Tyto dva prvky se nedají přímo srovnat, jelikož task při vykonávání zadané činnosti využívá vlákna. Rozdílem je, že při přímém používání vláken programátor musí zajistit jejich správné nastavení, spuštění a další. V případě použití tasků veškerou činnost okolo vláken zajišťuje TPL namísto programátora. Pod správou TPL tak může například několik tasků využívat dle potřeby jedno a nebo více vláken. TPL tak rozděluje a řídí činnost všech tasků s ohledem na aktuálně dostupný hardware.

O počet vláken, která budou vytvořena a o jejich následnou správu, se stará systémový správce vláken. Tento správce existoval již v dřívějších verzích .NET a jeho hlavním cílem bylo částečně zjednodušit a zefektivnit práci s vlákny tím, že nabízela, dle potřeby programátora, vlákna pro vykonávání požadovaných asynchronních operací. Nejnovější verze .NET je navíc doplněna o funkce, které na základě vytížení systému a dalších parametrech vytvoří potřebný počet vláken pro požadované tasky s cílem co nejefektivnějšího využití dostupných výpočetních zdrojů.

Další nově uvedenou technologií pro efektivnější vývoj paralelních aplikací v .NET je Parallel LINQ (PLINQ). LINQ (Language Integrated Query) je dotazovací integrovaný jazyk. Jeho hlavním účelem je zjednodušená práce s daty. Umožňuje ve

vybraných datech snadno hledat, třídit je atd. Existuje několik verzí LINQ podle typu dat, se kterými pracuje. Verzi, kterou PLINQ paralelizuje se nazývá LINQ to Object. LINQ to Object pracuje s výčtovými typy (IEnumerable) dostupnými v .NET. PLINQ, obsahuje kompletní sadu funkcí jako LINQ a navíc je doplňuje o několik metod, umožňujících jejich paralelní využití. PLINQ tedy pracuje na první pohled stejným způsobem jako jeho sekvenční alternativa LINQ, avšak nová verze se snaží při vykonávání zadaných operací využít všechna dostupná jádra procesoru tím, že rozdělí vstupní data na několik částí a provádí tak činnost na více částech současně, je-li to možné.

Nové jsou také některé datové struktury vytvořené speciálně pro tvorbu paralelních aplikací. Tyto struktury je možné využívat, jak s již výše uvedenými technologiemi TPL nebo PLINQ, tak i s kteroukoliv jinou vícevláknovou metodou. Jsou tedy takzvaně thread-safe. Obsahují různé nízko úrovňové synchronizační metody, které umožňují práci se strukturami z několika vláken současně bez obavy z možných souběhů (hazardů). Díky nízkoúrovňovým metodám synchronizace, které jsou v těchto strukturách implementovány, by měly dosahovat při jejich použití vyšší výkonnosti než v případě vlastnoručně vytvořených synchronizačních metod v kombinaci se standardními datovými strukturami.

2.3 Konkurence

Jedním z problémů, který nastává při tvorbě paralelních aplikací, je sdílení společných prostředků.

Možným řešením je například synchronizace. Každá z dostupných synchronizačních metod určitým dílem negativně ovlivňuje výkonnost. Kód obsahující synchronizační prvky bude i v případech kdy je spuštěno jen jedno vlákno, pomalejší než čistě sekvenční kód bez synchronizace. Z toho důvodu je nutné pro dobrou výkonnost použít jen nutné množství synchronizačních prvků a přizpůsobit kód tomu, aby bylo nutné synchronizovat co nejméně, avšak zajistit, aby nemohlo dojít k žádným hazardům na sdílených objektech. Synchronizačních metod v .NET existuje několik. Jejich výběr záleží na způsobech jakým se konkrétní sdílená data používají.

Nejjednodušší způsob je zámek (lock), který se aplikuje na globální objekt. Jeho funkcí je serializace přístupu k oblasti, kterou je nutné synchronizovat. Vlákno, které uzamkne zámek jako první, má výhradní právo pro přístup k datům a ostatní vlákna musí čekat na uvolnění zámku. Metoda zámku používá některé členy z třídy Monitor, konkrétně Enter a Exist, a vývojář má možnost, se zachováním stejné funkčnosti, použít i ty. Lock tedy existuje hlavně pro usnadnění a zpřehlednění práce při tvorbě programu.

Další metodou přístupu ke společnému objektu je použití vzájemného vyloučení (Mutual exclusion, Mutex). Jde o metodu, kdy se Mutexem uzamkne část kódu, ve které se nachází společný objekt. Po zamčení má přístup k objektu opět pouze vlákno, kterého uzamklo a ostatní vlákna musí čekat na jeho opětovné odemknutí.

Obě popsané metody se na první pohled zdají dosti podobné, avšak metoda lock vychází z .NET a Mutex je přímo z WIN32 a je tak na jiné úrovni. Prostřednictvím Mutexu je možné synchronizovat i mezi různými aplikacemi v rámci operačního systému.

Na první pohled je z uvedených metod zřejmý hned první jev, který ovlivňuje výkonnost. Při zamčení má k zamčenému objektu přístup vždy výhradně jedno vlákno, ostatní vlákna musí čekat. Druhý problém se projeví s dobou, kterou vlákna tráví v uzamčené části. Tím problémem je následná špatná škálovatelnost aplikace. Pokud například ukládá více vláken data do jednoho souboru, rychlost je ovlivněna rychlostí disku a dalších částí. V případě, kdy takové vlákno tráví například 1/25 svého času

v uzamčené oblasti, je možné využít v ideálním případě maximálně 25 vláken, při větším počtu už nebude aplikace v této oblasti dále škálovatelná a přestane tedy růst výkonnost aplikace s počtem vláken.

Upravenou verzí metody zámku (lock) je nová metoda ReaderWriterLockSlim. V dřívějších verzích .NET byla obdobná metoda s názvem ReaderWriter. Jednalo se však o složitý, pomalý (a těžký) systém synchronizace a proto do nové verze .NET Microsoft vytvořil odlehčenou a určitým způsobem optimalizovanou verzi ReaderWriterLockSlim a doporučuje na dále používat pouze tuto novou verzi, která nabízí vyšší výkonnost, jednodušší a přehlednější možnosti použití pro vývojáře. Tato metoda synchronizace sleduje způsob, jakým k datům jednotlivá vlákna přistupují a v případech, kdy nehrozí možnost hazardu jako například v případech čtení, umožní přistoupit k datům i více vláknům současně. Pokud se ale některé vlákno chystá data zapisovat, ostatní musí čekat až svou činnost dokončí. Uvnitř tedy pravděpodobně metoda obsahuje něco na způsob dvou zámků, jednoho pro čtení a druhý pro zápis, s tím, že čtecí zámek může zamknout i více „čtenářů“.

Druhým možným způsobem je asynchronní přístup k objektům. V takovém případě vlákna, vykonávající pracovní činnost, předají data, určená například k uložení, asynchronní metodě vytvořené pouze za účelem ukládání. Tak nejsou všechna pracovní vlákna zdržována a blokována zdlouhavým procesem ukládání, ale pouze předají data v paměti jinému procesu, který zpracovává obdržená data dle potřeby. V takovém případě nejsou data zpracována okamžitě, ale až když na ně přijde řada. Tento způsob je sice možné použít i v mé aplikaci, ale pouze v některých případech kdy, není důležitý přesný okamžik zpracování, jako například ukládání záložních dat do souborů.

3 Testování výkonnosti, profilování

3.1 Výkon (Performance)

Výkon nebo výkonnost aplikace se dá definovat mnoha různými parametry. Ve většině případů záleží na typu aplikace. U každé aplikace jsou důležité jiné vlastnosti a každá je vytvořená za jiným účelem. A tak i výkon je představován různě. Některé parametry, definující výkonnost, jsou však pro většinu aplikací společné. Těmi jsou například využití výpočetního výkonu procesoru nebo využití operační paměti. Dalším parametrem, vyjadřující charakter výkonu, může být sledování doby potřebné pro vykonání úkolů, které má aplikace v popisu činnosti. V souhrnu je možné říci, že je vždy dobré sledovat průběh primární činnosti aplikace, nebo-li činnosti, při které zvolená aplikace tráví nejvíce času. V takovýchto oblastech je většinou možné docílit největších zisků z hlediska její celkové výkonnosti.

3.1.1 Profillery

Měření stanovených parametrů je možné provádět několika různými způsoby. Existuje řada různých aplikací, které umožňují testování a měření výkonu. Tyto aplikace jsou všeobecně nazývány jako profillery. Jejich úkolem je sledovat uživatelem nastavené parametry. Některé pokročilejší jsou schopné sledovat i spouštění jednotlivých metod přímo v testované aplikaci a měřit dobu jejich trvání. Výstupem takové analýzy může být seznam všech spuštěných metod, spolu s dobou jejich celkového trvání a také například strom provázanosti jednotlivých metod mezi sebou. Takovéto funkce nabízí například profiller integrovaný ve Visual Studiu. A tedy v případech, kdy je aplikace vytvářena v prostředí Visual Studia, není nutné používat některou z dostupných externích aplikací, jelikož i samotné Visual Studio obsahuje nástroje umožňující měření výkonu, sledování zdrojů, využití paměti a další. Zmíněné nástroje jsou seskupeny v modulu Performance Wizard.

3.1.2 Výkonnostní čítače

Dalším možným způsobem zisku dat pro měření výkonnosti jsou výkonnostní čítače (performance counters). Všeobecně je možné výkonnostní čítače dělit na dvě skupiny. Systémové, které jsou nabízeny automaticky systémem, a vlastní, které si

vývojář softwaru může vytvořit sám. Vlastnoručně vytvořené čítače je nutné integrovat přímo na vhodná místa v kódu vytvářeného programu, dle jeho struktury a také dle požadavků na funkci, kterou má daný čítač plnit.

Systémových čítačů je v OS Windows velké množství. Disponuje řadou od čítačů sledujících využití pevného disku, procesoru, stavu sítě až po sledování tiskových front k tiskárnám. Nejzajímavější skupinou čítačů z mého pohledu je skupina *Process*. Tato skupina disponuje čítači, kterými jsou vybaveny všechny běžící aplikace a systémové procesy. I v této skupině je celá řada různých výkonnostních čítačů. Mezi ty z mého pohledu nejdůležitější patří například čítač využití operační paměti sledovaným procesem nebo počet vláken, které daná aplikace momentálně využívá. Zajímavým čítačem je čítač sledující množství dat, která do aplikace vstoupila, ať už formou přečtení dat ze souboru, přes síť nebo dalšími možnými způsoby.

Vlastní výkonnostní čítače se integrují přímo do kódu sledované aplikace. Mohou plnit řadu různých funkcí dle požadavků programátora. Ať už měření doby trvání vybraných činností, nebo sledování počtu přijatých zpráv či počet přijatých zpráv za sekundu. Vytvořené čítače musí být zařazeny do vlastní skupiny, která je sdružuje. Skupina je zde pro umožnění sledování hodnot čítačů za běhu vybrané aplikace, kdy pro možnost jejich monitorování je nutné nové čítače spolu s jejich skupinou zaregistrovat do systému. Po zaregistrování jsou hodnoty těchto čítačů přístupné pro aplikace sloužící k jejich monitorování.

Stav a hodnoty jednotlivých čítačů, ať už vlastních nebo systémových, je možné sledovat a případně i zaznamenávat různými aplikacemi. Tím nejjednodušším způsobem je sledovat hodnoty prostřednictvím nástroje *Sledování výkonu*, který se nachází ve správci systému. Tento nástroj umožňuje sledovat veškeré čítače, které se v době jeho spuštění nacházejí v systému, kromě sledování aktuálně získaných dat je schopný i měřená data ukládat do nastaveného souboru. V mém případě jsem použil k záznamu hodnot čítačů již výše zmíněný profiller, integrovaný v prostředí Visual Studia. Nevýhodou čítačů, která se projevuje hlavně při sledování rychlejších dějů, je velká perioda aktualizace jejich hodnoty. Nejvyšší rychlost a tedy nejmenší perioda jakou jsou čítače aktualizovány je pouze 500ms.

3.2 Spolehlivost (Reliability)

Spolehlivost jsem si definoval jako schopnost aplikace vykonávat svoji zadanou činnost v ostrém provozu. V případě systémové služby, kterou má aplikace je, musí být schopna nepřetržitého dlouhodobého provozu. Spolu s nepřetržitým chodem musí služba také nepřetržitě a spolehlivě vykonávat zadané činnosti. V případě neočekávané chyby musí být aplikace také schopna se z tohoto stavu zotavit a po té dále pokračovat v požadované činnosti. Spolehlivost je možné testovat sledováním jejího chování v provozu, ale ani tak není možné odhalit všechny problémy. Vždy tedy existuje určitá nejistota, že může dojít k problému. Aplikace tedy musí být vybavena důmyslným systémem zachytávání vzniklých neočekávaných chyb, který má za úkol zabránit jejímu pádu za každé okolnosti. Tento systém musí být o to složitější v případě, kdy aplikace pracuje s více vlákny současně.

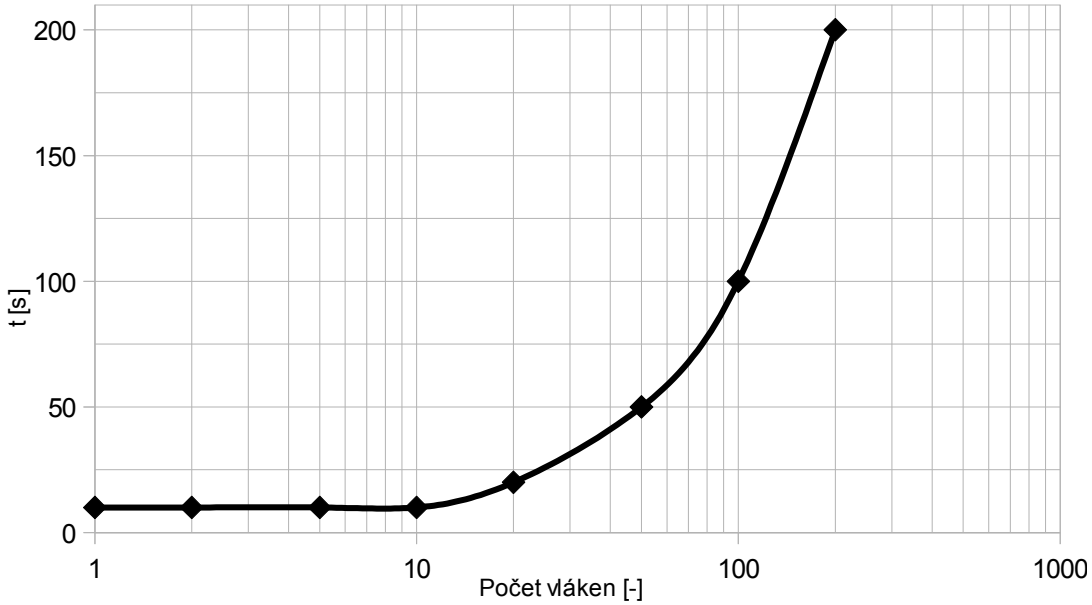
3.3 Škálovatelnost (Scalability)

Dobrá škálovatelnost aplikace je důležitá u aplikací pracujících s více vlákny. Škálovatelností je specifikováno do jaké míry se s rostoucím počtem vláken bude zvyšovat výkonnost aplikace. V ideálním případě se zadaná úloha vykoná n-krát rychleji při použití n-vláken oproti zpracování úlohy vláknem jedním. To vše v případě dostatečných hardwarových zdrojů. V praxi však takového výsledku ani při použití dostatečně výkoného hardwaru není možné docílit. Vícevláknové zpracování by tak mělo nabízet co nejvyšší výkonnost v rámci možností.

Činitelů, kteří více či méně ovlivňují schopnost škálovatelnosti aplikace je celá řada. Těmi hlavními a dá se říci i nejvýznamnějšími v podobných úlohách jsou způsoby synchronizace použité pro sdílení společných dat v rámci jednotlivých vláken aplikace. Každá synchronizace sice sníží schopnost aplikace škálovat, ale ne vždy tím musí vzniknou vážný problém. Záleží totiž na potřebách konkrétní aplikace. I přesto, že dojde ke snížení škálovatelnosti aplikace, nemusí dojít ke snížení v takové míře, které by bylo pozorovatelné v běžném provozu. Proto je nejprve nutné si ujasnit jak bude praktický provoz vypadat a zda je současná úroveň škálovatelnosti pro něj dostačující.

Škálovatelnost se projevuje dvěma hlavními způsoby. V případech, kdy je zadán úkol, který se dá zpracovávat paralelně, se limit škálovatelnosti projeví tím, že další paralelizací už dále nebude docházet ke snižování doby, za kterou se požadovaný úkol

vykoná. Druhým typem jsou úkony, při kterých aplikace zpracovává mnoho různých úloh s některými sdílenými prvky. Těmi mohou být sdílená data, komunikační kanál a podobně. V těchto případech se omezení další škálovatelnosti aplikace projeví tím, že s nárůstem úkolů začne docházet ke zpomalení ve zpracování ostatních úloh i v případech, kdy je k dispozici dostatek výpočetních zdrojů. Tento případ je znázorněn na následujícím grafu (viz. Graf 3.1), kde je zřejmé, že měřená funkce je škálovatelná maximálně pro 10 vláken a s dalším rostoucím počtem vláken dochází k výraznému zpomalení. Dle podoby a chování mé aplikace jsem ke škálovatelnosti přistupoval z druhého jmenovaného pohledu, tedy z pohledu sdílení společných dat mezi vlákny.



Graf 3.1: Demonstrace škálovatelnosti – Úkolem vláken v demonstrační aplikaci bylo spustit zadanou funkci, která pracovala přibližně 10% svého času se sdílenými daty a následně s načtenými hodnotami prováděla jednoduché matematické úkony

4 Popis aplikace

4.1 Podoba současné aplikace

Jedná se o systémovou službu pracující v operačních systémech Windows s platformou .NET 2.0 vytvořenou v programovacím jazyce C#. Jelikož systémové služby nemají žádné grafické rozhraní (GUI), komunikace s uživatelem je realizována prostřednictvím grafické (klientské) aplikace. Spojení mezi systémovou službou a klientskou aplikací je vytvořeno prostřednictvím mezi-procesové komunikace (IPC). Existuje řada různých druhů mezi-procesové komunikace, v případě mé aplikace je použita technologie Microsoft Message Queuing (MSMQ). Tato technologie vytváří v operačním systému fronty zpráv, prostřednictvím kterých systémová služba komunikuje s grafickou aplikací.

Hlavní náplní systémové služby je sběr dat z měřících přístrojů společnosti KMB systems, s.r.o.. Sběr dat se skládá z periodického stahování aktuálně měřených dat nebo archivních záznamů a jejich následného ukládání do zvolených databází. Systémová služba je tedy spojena také s SQL databázemi prostřednictvím objektově-relačního mapování (ORM) rozhraní XPO. XPO je prostředek spojující prvky databáze s objektově orientovaným prostředím programovacího jazyka C#.

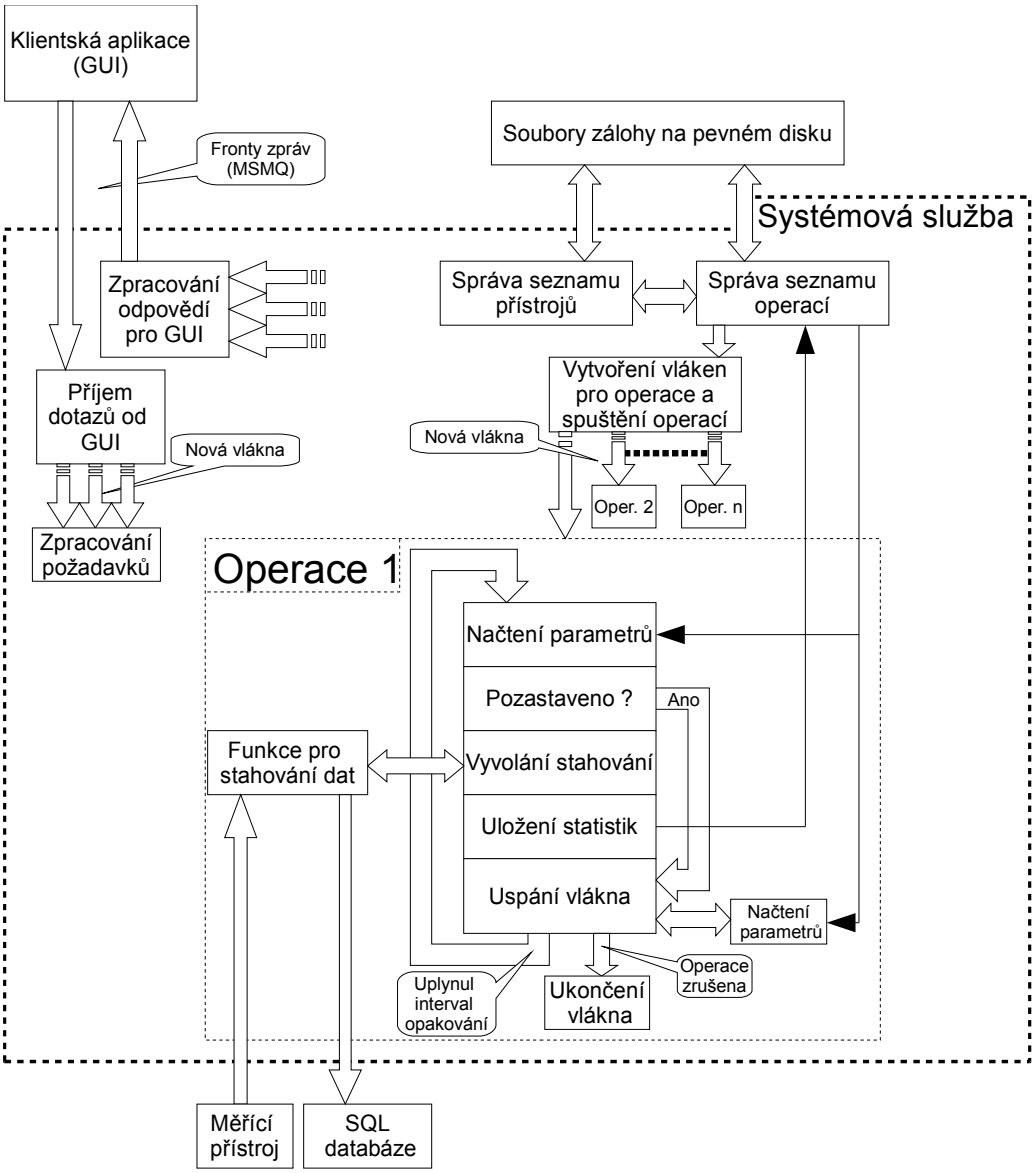
Každý požadavek, který uživatel systémové službě zadá, je vykonáván v novém vlákně. Je tedy schopná vykonávat paralelně více úkolů současně. Část vláken je vytvářena automaticky prostřednictvím asynchronního příjmu zpráv pomocí fronty zpráv technologie MSMQ. Jak jsem již dříve zmínil hlavní prací systémové služby je cyklické stahování měřených dat z měřících přístrojů. Pro tuto činnost jsou v aplikaci vytvářena vlákna programově, dle počtu a potřeby jednotlivých operací. Každá tato periodická operace má pro svůj běh v aplikaci vyhrazeno vlastní vlákno. Ve vytvořených vláknech následně jednotlivé operace vykonávají uživatelem zadanou činnost. Nová vlákna pro operace, které periodicky stahují data, jsou vytvářena, buď při startu systémové služby, což platí pro již existující operace, jejichž parametry jsou načteny ze souboru. Druhým způsobem jsou případy, kdy uživatel zadá prostřednictvím klientské aplikace systémové službě parametry pro vytvoření nové operace. V tomto případě je vytvořeno nové vlákno, které náleží nově vzniklé operaci ihned po jejím zaregistrování do vnitřního seznamu, který systémová služba obsahuje.

Jak jsem zmínil, každá zadaná operace má své parametry, které specifikují její činnost. Mezi parametry patří například jméno databáze, druh dat, která mají být stažena a další. Tím nejdůležitějším parametrem ale je perioda opakování. Tato hodnota udává v jakých intervalech bude příslušná operace opakována. Tento interval se vždy měří od dokončení předchozího běhu náležité operace. Všechny parametry jsou ve formě struktury pro každou operaci uloženy v již zmíněném seznamu, který je sdílený ostatními operacemi. Jelikož je tento seznam sdílený a je možné k němu v rámci aplikace přistupovat z více vláken současně je přístup k němu synchronizován pomocí zámku, který zabraňuje možným hazardům. Zámek dovolí přístup k seznamu a veškerou práci s ním vždy pouze jednomu vláknu v jeden okamžik. V případě uzamčení zámku ostatní vlákna musí počkat do jeho uvolnění, kdy bude dalšímu vláknu umožněn přístup k seznamu.

Průběh každého cyklu operace začíná načtením parametrů, specifikujících konkrétní operaci ze seznamu. Protože uživatel má možnost kdykoliv prostřednictvím klientské aplikace upravit parametry operace, je nutné je při každém cyklu nově načíst, aby byla zajištěna jejich aktuálnost. U periodických úkonů má uživatel možnost nejen upravit jednotlivé parametry, ale může také jednotlivé úkony pozastavit. Z toho důvodu je nutné po načtení parametrů ověřit, zda není konkrétní operace pozastavena, v takovém případě se přeskočí stahování dat a čeká se na opětovné spuštění vyvolané uživatelem. Následně po ověření, že operace není pozastavena a že jsou všechny parametry platné, dojde k vyvolání funkce z knihoven KMB, která provede stažení požadovaných dat a jejich uložení do databáze, přičemž se vše řídí dříve načtenými a zadanými parametry. Po stažení požadovaných dat z přístroje a jejich uložení do databáze dojde k uložení statistik o průběhu stahování. Statistická data jsou ukládána do stejného seznamu, v jakém jsou uloženy parametry jednotlivých operací. Ke každé operaci, nacházející se v tomto seznamu, přísluší struktura obsahující její statistické informace. Tyto informace se skládají z čítačů, které počítají počet úspěšných a neúspěšných stažení dat, dále uchovávají datum a čas posledního pokusu o stažení. Nakonec je tam také v případě neúspěchu zahrnut důvod, kvůli kterému operace selhala. Všechna statistická data se uchovávají pro potřeby uživatele. O jejich hodnotách je informován jak v grafické aplikaci, přes kterou komunikuje se systémovou službou, tak i prostřednictvím emailových reportů, které může uživatel v případě zájmu přijímat. Po stažení dat a uložení statistických informací je vlákno, ve kterém se operace vykonává,

uspáno. Vlákno je uspáno na dobu, kterou představuje interval opakování operace nebo do doby, kdy jsou parametry operace změněny nebo je operace uživatelem odebrána. Pokud je tato hodnota 0, operace je ukončena. Nejnižší platná hodnota intervalu je 1s. Uspané vlákno je v pevně nastavených intervalech probouzeno, aby mohlo ověřit, že nedošlo k odebrání příslušné operace a také zda nebyly upraveny její parametry. V tomto případě je nejdůležitějším parametrem interval opakování, při jehož změně se musí změnit i doba, na kterou je vlákno uspáno.

Všechna vlákna, která zastávají zadané operace v systémové službě, existují po celou dobu, kdy je systémová služba spuštěna nebo do doby, než se uživatel prostřednictvím klientské aplikace rozhodne danou operaci odebrat.



Obr. 4.1: Struktura současné aplikace

5 Analýza a měření parametrů

5.1 Parametry k měření

Před započetím veškerých testů je nejprve nutné stanovit v aplikaci oblasti, které je vhodné testovat. Dále se musí určit druhy testů a způsoby měření, které se budou na aplikaci provádět. Tyto věci není možné určit žádným způsobem univerzálně. Jelikož od každé aplikace se očekává jiné chování a jsou vytvořeny za jiným účelem, musí se ve většině případů lišit i druh a způsob prováděných testů výkonnosti. Například u aplikace, která má být v provozu dlouhodobě bez přerušení, jakou může být systémová služba, se očekávají výsledky v jiných oblastech než u grafické aplikace, která má co nejrychleji reagovat na příkazy uživatele a podobně.

Při analýze současné aplikace, před její úpravou a také upravené aplikace jsem sledoval řadu různých parametrů, které jsem měřil při různých testech. Různými testy mám na mysli různé úrovně zatížení systémové služby. Různého zatížení jsem dociloval počtem obsluhovaných přístrojů a jím příslušných operací, stahujících data. Při těchto testech jsem měřil využití soukromé paměti systémovou službou. Velikost soukromé paměti reprezentuje velikost alokované operační paměti, kterou nemůže využít žádný jiný proces nebo aplikace v systému. Další měřenou veličinou byla úroveň využití procesoru, představující využití procesoru systémovou službou v procentech. Posledním významnějším měřeným parametrem bylo měření doby potřebné k načítání všech potřebných konfiguračních parametrů pro jednotlivé běžící operace a také pro načtení seznamu všech obsluhovaných přístrojů spolu s jejich parametry.

5.2 Problémy odhalené při analýze

Současná struktura aplikace je založena na multivláknové činnosti. Pro každý úkol, který má systémová služba vykonat, je založeno nové vlákno. Životnost vlákna je omezena na dobu, po kterou je zadaný úkol vykonáván, po jeho dokončení vlákno zanikne. Avšak značné množství činností, pro které je primárně systémová služba určena, se vykonává cyklicky v předem nastaveném intervalu. V mezidobí jsou tato vlákna uspána a pouze čekají na uplynutí nastaveného intervalu. U většiny takových vláken, které vykonávají cyklicky opakované operace, se tak projevil značný nepoměr mezi jejich činností a nečinností. Každé operaci tedy bylo přiřazeno jedno vlákno a od

té doby se každá operace starala dále pouze sama o sebe. Operace si tedy musely načítat ze sdílených objektů parametry potřebné k jejich chodu i v případech, kdy čekaly na další cykl stahování. Toto chování vyvíjelo další zátěž na systém. Navíc každé takové spuštěné vlákno spotřebovává určité systémové zdroje. Alokuje si pro sebe část operační paměti v řádu několika MB. Dle testů dosahuje paměťová náročnost nového vlákna v průměru hodnoty 1-2 MB. Paměť tak byla zaplněna vlivem řady zbytečně žijících vláken. V době, kdy vlákna spí, pouze obnovují hodnoty konfiguračních parametrů a jsou tak zbytečná.

Z těchto důvodů jsou uspaná a tím nečinná vlákna vysoce neefektivní a jsou tak jedním z hlavních cílů pro optimalizaci a zvýšení výkonnosti aplikace. Dalším problémem, který je s tím spojen, je časté načítání konfiguračních hodnot ze seznamů, které jsou sdíleny mezi všemi vlákny. Tím dochází k vysoké konkurenci mezi nimi a tak i ke zbytečnému zdržování, způsobeného synchronizací přístupu ke zmíněným datům.

5.3 Návrh optimalizované podoby aplikace

Vzhledem k tomu, že největší část svého pracovního času je systémová služba zaměstnána cyklickým stahováním dat z měřících přístrojů, zaměřil jsem se tedy zejména na optimalizaci částí služby, které přímo souvisí s periodickými operacemi. Změny se tak týkají hlavně správy seznamu operací a podoby operací samotných, ale dotkli, se také podoby správy seznamu přístrojů a to hlavně způsobem synchronizace přístupu ke společným datům. Způsobů, jakým je možné upravit podobu a strukturu kritických částí aplikace, je více. Proto jsem se rozhodl navrhnout více možností, kterými by se dala struktura vylepšit, a až na základě praktických testů zvolit tu nejvhodnější podobu.

Za jeden z primárních cílů, při návrhu nové struktury, jsem si dal odstranění dlouhožijících vláken, ve kterých se provádějí cyklické operace. Převážnou dobu svého života jsou tyto operace uspané. Záleží sice na konkrétním typu přístroje a jeho nastavení a také na velikosti periody opakování, ale ve většině případů trvá stahování potřebných dat v jednom cyklu řádově několik sekund. Intervaly opakování se však v praktickém nasazení pohybují v řádech desítek minut až hodin a v některých případech mohou dosahovat i jednotek dnů. Je tedy značně neefektivní, vzhledem k systému, udržovat v aplikaci takováto dlouhožijící spící vlákna, která svojí činnost

vykonají za zlomek času, po který jsou spuštěny. Rozhodl jsem se tedy do nových návrhů zahrnout systém, který by zajišťoval spouštění vláken s operacemi jen v případě potřeby, nemuselo by tak díky tomu docházet ke zbytečnému plýtvání systémových zdrojů vzniklé dlouhožijícími vlákny.

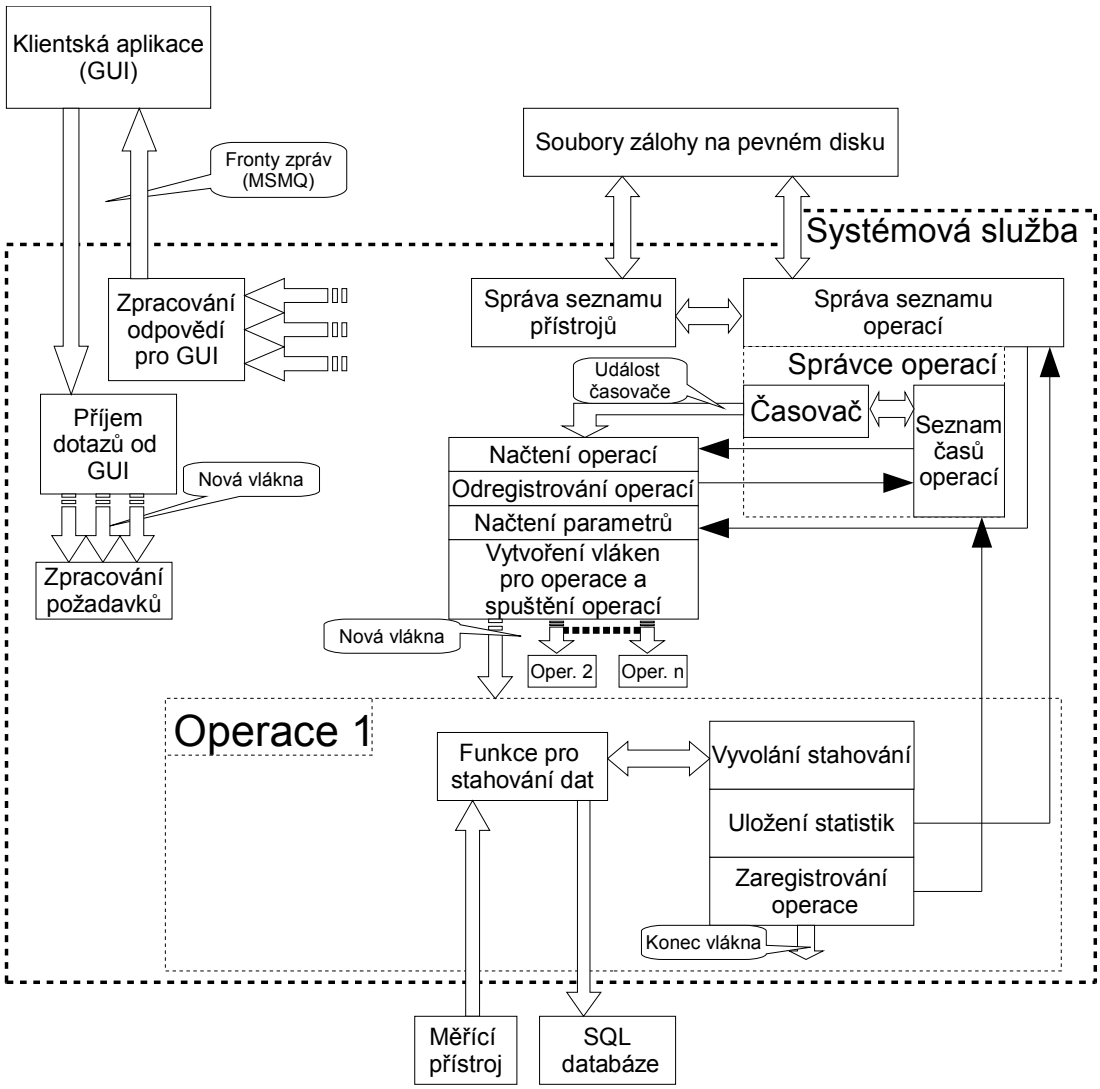
5.3.1 Řešení 1 – Centrální správce operací

V prvním řešení jsem se rozhodl vytvořit jakéhosi centrálního správce operací. Ten rozšiřuje a v některých případech nahrazuje současné funkce, které se nachází ve správě seznamu operací. Jeho základem je upravená verze časovače, který se nachází v systémové knihovně *Timers*. Úpravy, které jsem provedl, se týkají hlavně přizpůsobení vlastností časovače vícevláknovému provozu, kterým systémová služba disponuje. Vytvořený časovač nabízí pouze nejnutnější funkce potřebné k jeho použití. Těmi funkcemi je možnost přidělení události, která bude vyvolána po uplynutí intervalu, dále spuštění a zastavení časovače a samozřejmostí je také nastavení intervalu. Časovač dále automaticky pracuje v režimu, při kterém se po uplynutí nastaveného intervalu a vyvolání přiřazené události sám zastaví. Úpravy umožňující snadný a bezproblémový vícevláknový provoz se týkají hlavně použití vhodných synchronizačních technik. Ty se skládají převážně z metody zámků. Zámky zabraňují hazardům, které by mohly vznikat při současném přístupu více vláken k jednomu objektu. Hazardům je zabráněno uzamčením objektu proti přístupu z ostatních vláken, vyjma vlákna, které má zámek v držení.

Správce operací dále, vyjma časovače, zahrnuje seznam, který obsahuje přesné časy, ve kterých mají být jednotlivé operace spuštěny. Identifikace jednotlivých operací je zajištěna pomocí unikátních identifikačních čísel, která jsou operacím přiřazena při jejich zadání uživatelem. V tomto seznamu jsou časy stahování jednotlivých operací řazeny vzestupně tak, že časově nejbližší operace je na prvním místě. Při plnění seznamu časů je současně nastaven interval časovače a také událost, která bude vyvolána po uplynutí intervalu. Hned po té je časovač odstartován. Po uplynutí intervalu dojde k současnému spuštění nastavené události a automatickému zastavení časovače. Událost je nastavena tak, aby vyvolala metodu, která zajistí další kroky vedoucí ke spuštění operací, kterým odpovídá nastavený čas.

Spuštěná metoda vyjme ze seznamu časových hodnot veškeré operace, jejichž

čas již uplynul, dále je nastaven práh, který zajistí vyjmutí i případných operací, které by měli být spuštěny v blízké budoucnosti. Rozlišení časovače je v jednotkách milisekund, avšak pro veškeré operace postačuje rozlišení v sekundách. Z tohoto důvodu jsem doplnil při spouštění operací zmíněnou prahovou hodnotu, která činí prozatím 500ms. Práh zajišťuje, aby nedocházelo k příliš častému a „zbytečnému“ spouštění časovače, vzhledem k ne až tak kritickým nárokům na přesnou dobu spouštění operací. Tím je zajištěno, že v nejhorším a ne moc pravděpodobném případě se časovač spustí každých 500ms. Po vyjmutí všech operací, které budou následně spuštěny, ze seznamu časů, dojde k nastavení intervalu časovače na další nejbližší operaci v seznamu a k jeho spuštění. Po té jsou podle identifikačních čísel vyjmutých operací načteny jejich parametry pro stahování. Po načtení veškerých parametrů pro jednotlivé operace už dojde k fyzickému spuštění samotných operací obdobně jako tomu bylo v současné verzi. Avšak samotný průběh činnosti ve vlákně, ve kterém se vykonává operace, se díky centrálnímu správci operací značně zjednodušil. Protože jsou již načteny parametry potřebné ke stahování dat, dochází ihned po spuštění vlákna s operací ke spuštění funkce, která zajišťuje stahování dat z přístroje a jejich následné ukládání do databáze. Po dokončení stahování jsou obdobně jako u předchozí verze uloženy statistické informace o proběhlém stahování. Po dokončení všech úkonů dojde k oznámení o ukončení stahování a k zaregistrování dokončené operace zpět do seznamu časů ve správci operací. Čas dalšího spuštění, který se do seznamu ukládá, je určen na základě aktuálního času a intervalu opakování. Po úspěšném zaregistrování příslušné operace je vlákno ukončeno. Ověřování, zda náhodou uživatel nepozastavil některou operaci nebo zda ji neodebral, se v tomto řešení eliminovala. Při vyvolání příkazu k odebrání nebo pozastavení operace dochází prostřednictvím centrálního správce k odregistrování dané operace ze seznamu. Při opětovném odstartování pozastavené operace je samozřejmě operace okamžitě zaregistrována.

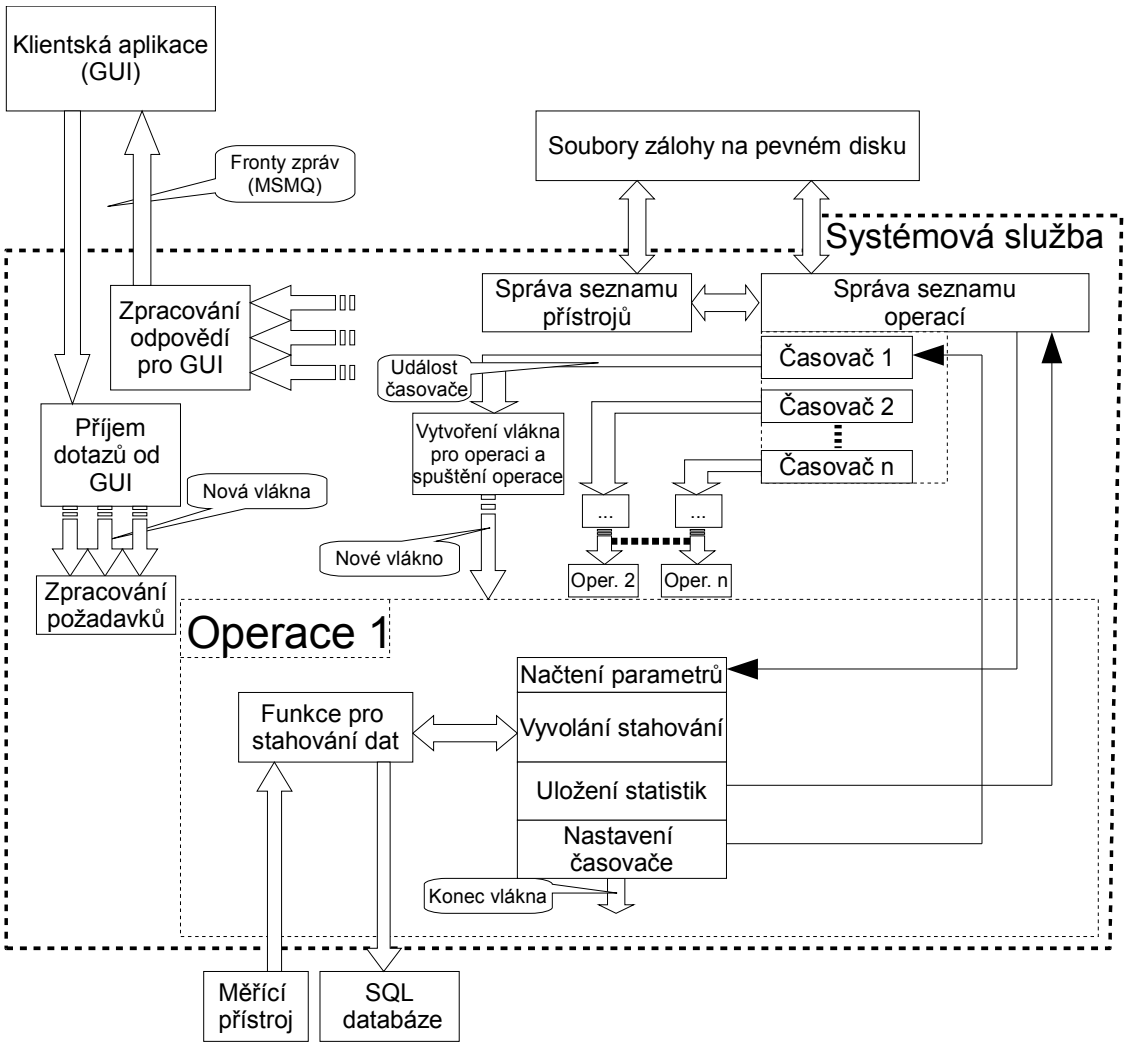


Obr. 5.1: Návrh nové struktury – Centrální správce operací

5.3.2 Řešení 2 – Decentralizovaný systém s časovači

V tomto řešení jsem se rozhodl přiřadit více priority objektům (strukturám), obsahujícím parametry jednotlivých operací. Tyto objekty jsou uloženy v seznamu operací. Tyto struktury, tvořící již zmiňovaný objekt, obsahující parametry přiřazené operace, jsem rozšířil o několik metod schopných řídit operaci. Každá operace tak má vlastní soukromou správu, která se stará o její spouštění, časování a vytváření nových vláken. Jedná se tedy o jakousi decentralizovanou správu operací. Spouštěcím prvkem pro každou operaci je její vlastní časovač. Oproti předchozímu řešení, kde byl pouze jeden centrální časovač, zde má každá operace přiřazen vlastní. Vzhledem k tomu, že pro většinu funkcí, týkajících se nějakým způsobem chodu operací, platí pravidlo, co

jedna spuštěná operace, to jedno vlákno. Z toho důvodu nebylo třeba řešit problém synchronizace u některých nových částí. Tím pádem odpadla i nutnost vytvářet vlastní upravenou verzi časovače a použil jsem tedy přímo standardní systémový časovač z knihovny *System.Timers*. Nastavení časovačů je obdobné jako v předchozím řešení. Tedy, že po uplynutí nastaveného intervalu a vyvolání nastavené události dojde k jejich automatickému zastavení. Tím, že každá operace má vlastní časovač s vlastní událostí, každá zvlášť si i načítá potřebné parametry pro stahování dat. V tomto řešení událost vyvolaná časovačem načte parametry a přímo spustí nové vlákno, určené pro stahování dat příslušné operace. Odpadá tak i potřeba dvou správců, kteří byli potřeba v předchozím řešení. Jeden, který zajišťoval spouštění časovače a tím i příslušných operací a druhý, který se „fyzicky“ stará o jednotlivé operace prostřednictvím jejich parametrů. Každá operace se tak řídí nezávisle na těch ostatních. Toto řešení nabízí značnou redukci kódu, kdy není potřeba využívat větší množství složitějších funkcí na úkor navýšení počtu časovačů. Počet časovačů se nyní rovná počtu operací, což je značný nárůst oproti předchozímu řešení, kde byl použit pouze jeden upravený. Ani v tomto řešení nedochází ke zbytečnému uspávání vláken v dobách, kdy je některá z operací pozastavena nebo když čeká na čas dalšího stahovacího cyklu. V případech, kdy je operace uživatelem pozastavena, dojde k jednoduchému zastavení příslušného časovače. Při odebrání zvolené operace dojde automaticky i k zastavení a odebrání časovače, jelikož se nachází ve stejné struktuře jako ostatní parametry, přiřazené k odebírané operaci a tvoří tak dohromady jeden objekt.



Obr. 5.2: Návrh nové struktury – Decentralizované řízení operací

6 Technická řešení nové struktury

6.1 Analýza možných metod synchronizace sdílených dat

V případech, ve kterých aplikace pracuje ve vícevláknovém provozu a jednotlivá vlákna sdílejí společná data, musí se zajistit, aby nedocházelo k datovým hazardům. Hazardy nastávají v případech, kdy se dvě a více vláken snaží v jeden okamžik upravit stejný objekt. Jejich důsledkem jsou předem neočekávané hodnoty sdílených dat. V následujícím jednoduchém příkladu (viz. Výpis 6.1) je znázorněno, co za následky takové hazardy mohou mít. Zvolil jsem jednoduchý příklad, ve kterém dvě vlákna pracují s jednou společnou proměnnou. Výchozí hodnotou sdílené proměnné je 0, jedno vlákno jí inkrementuje a druhé dekrementuje. Každé vlákno vykoná 1000000 iterací.

```
int X = 0;
iterations = 1000000;
Thread Thread1;
Thread Thread2;
for (int i = 1; i < 6; i++)
{
    X = 0;
    Thread1 = new Thread(() =>
    {
        for (int a = 0; a < iterations; a++)
        {
            X--;
        }
    });
    Thread2 = new Thread(() =>
    {
        for (int a = 0; a < iterations; a++)
        {
            X++;
        }
    });
    Thread1.Start();
    Thread2.Start();
    while (Thread1.IsAlive || Thread2.IsAlive)
    {
        Thread.Sleep(1);
    }
    Console.WriteLine("Test {0}: X = {1} ", i, X);
}
```

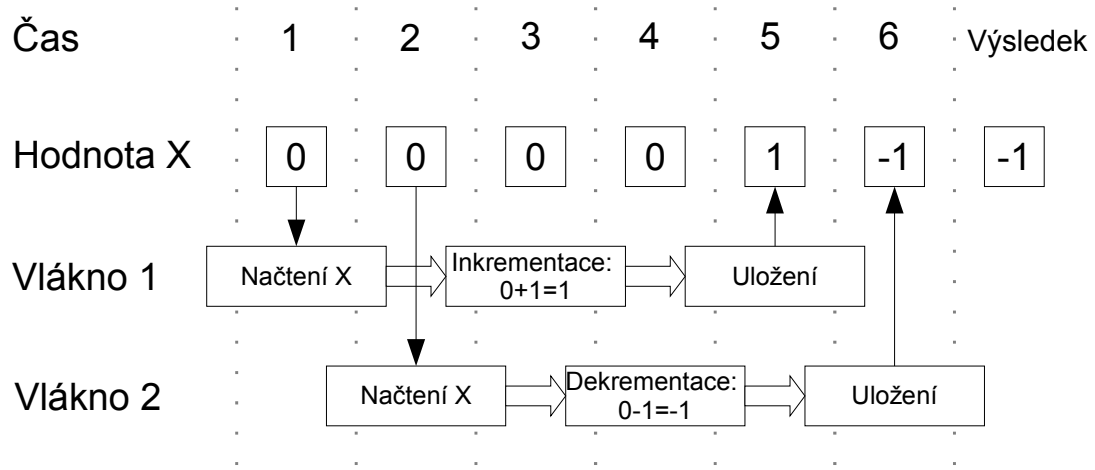
Výpis 6.1: Ukázka kódu s riziky hazardů

Předpokládanou výslednou hodnotou proměnné X je 0, jelikož obě vlákna mají stejný počet iterací a pouze opačné funkce. Avšak vlivem hazardů je výsledek značně nejistý. Z pěti provedených testů se ani jedna hodnota neshoduje s předpokladem a dokonce jsou rozdílné i vzájemně, viz. následující ukázka výstupu (viz. Výpis 6.2).

```
Test 1: X = -51573
Test 2: X = -514283
Test 3: X = -381
Test 4: X = -393819
Test 5: X = 359455
```

Výpis 6.2: Výstup z demonstračního programu

Důvodem proč k tomuto jevu dochází je, že jedna samotná úprava proměnné X, ať už její inkrementace nebo dekrementace, se vnitřně skládá ze tří kroků. Nejprve je z proměnné načtena její aktuální hodnota. Načtená hodnota je upravena (inkrementace, dekrementace) a nakonec je výsledná hodnota opět uložena do proměnné X. Problém následně nastává při zahrnutí druhého vlákna, protože i jeho činnost se skládá ze tří obdobných kroků. Obě vlákna nejsou synchronizována, takže nastávají případy, že vlákna načtou stejnou hodnotu a následně tak jedno přepíše upravený výsledek druhého. Vzniklou situaci popisuje následující obrázek (Obr. 6.1), který demonstruje jednu iteraci výše uvedeného kódu (viz. Výpis 6.1).



Obr. 6.1: Vznik hazardu

Výsledkem X z výše uvedeného obrázku je -1 i přes očekávanou hodnotu 0. Tyto problémy se musí eliminovat. A k tomu slouží synchronizace přístupu ke sdíleným datům.

V .NET je dostupných několik metod, které umožňují synchronizaci přístupu ke společným datům z více vláken. K těm nejznámějším způsobům patří metoda s názvem *lock*, neboli zámek. Vnitřně se jedná o metodu *Monitor* ze systémové knihovny *Threading,* kterou je také možné pro synchronizaci použít, ale dosahuje stejných výsledků jako *lock,* který existuje pro usnadnění práce programátora a zjednodušení synchronizace vláken ze strany programování. Není nutné psát tolik kódu jako při

použití přímo metody *Monitor*. Další možností je rozšířená verze *lock*, která rozlišuje čtení a zápis do sdílených dat. Její jméno je *reader-writter lock*. Nakonec je také možné použít *Mutex*, který je přístupný prostřednictvím knihovny *Threading*. Speciální metodou je také *SpinLock*, který využívá uspání vlákna pomocí *spinwait*. To znamená, že vlákno je sice v případě čekání na odemknutí zámku uspáno, ale výpočetní výkon není předán jiným neuspaným vláknům. Metoda by tak měla vykazovat dobré výsledky v případech, kdy se zamykání provádí na velmi krátké období. Vyjma synchronizačních metod je možné od verze .NET 4.0 použít také synchronizované struktury. Jde o struktury, které jsou již vnitřně vybaveny synchronizačními metodami a nehrozí tak nebezpečí vzniku hazardu při vícevláknovém provozu.

Není možné předem určit, která z metod vykazuje nejlepší výsledky, jelikož každá má své pro a proti. Proto jsem jednotlivé metody podrobil několika testům, které ukáží jejich výkonnost v rámci potřeb mé aplikace. Navržené metody testování se skládají ze tří hlavních částí, které se v praxi a hlavně v mé aplikaci vyskytují. Prvním je čistě čtení ze společných dat. Dále testuji kombinaci čtení a zápisu hodnoty v rámci sdílených dat. Nakonec testuji i schopnost synchronizace při zápisu. Některé metody nerozlišují, zda se chystá aplikace zapisovat nebo číst, ale jsou i takové, které tím rozlišením získávají na svém výkonu. Například metoda *reader-writer lock*. Z toho důvodu jsem test rozdělil na tři různé části, ve kterých se ukáží schopnosti jednotlivých metod.

V následujících výpisech jsou úryvky kódů popisující způsob jednotlivých testů.

```
public double GetValue_N(int i)
{
    double f = 0;
    for (int a = 0; a < 10; a++)
    {
        f += dict[i];
    }
    return f;
}
```

Výpis 6.3: Čtení ze sdílené struktury

Jako sdílený objekt s názvem *dict*, který je nutné synchronizovat, jsem použil slovník (*Dictionary*). Tuto volbu jsem provedl na základě jeho četného použití v aplikaci, kterou optimalizuji. V ukázce (viz. Výpis 6.3) dochází k načtení 10 hodnot, které vytváří umělé zpoždění, simulující například prohledávání struktury v ostrém provozu a podobně.

```
public double SetGetValue_N(int i)
{
    double f = 0;
    for (int a = 0; a < delay; a++)
    {
        f += dict[i];
    }
    dict[i] = new Random().NextDouble();
    return f;
}
```

Výpis 6.4: Čtení a zápis do sdílené struktury

Ve Výpisu 6.4 je znázorněno načítání hodnot ze sdíleného objektu v kombinaci s příležitostným zápisem. Tento test je zde hlavně pro synchronizační metody, u kterých je možné rozlišit, zda dochází ke čtení nebo zápisu dat.

```
public void SetValue_N(int i)
{
    double f = 0;
    for (int a = 0; a < 5; a++)
    {
        dict[i] = new Random().NextDouble();
    }
}
```

Výpis 6.5: Zápis do sdílené struktury

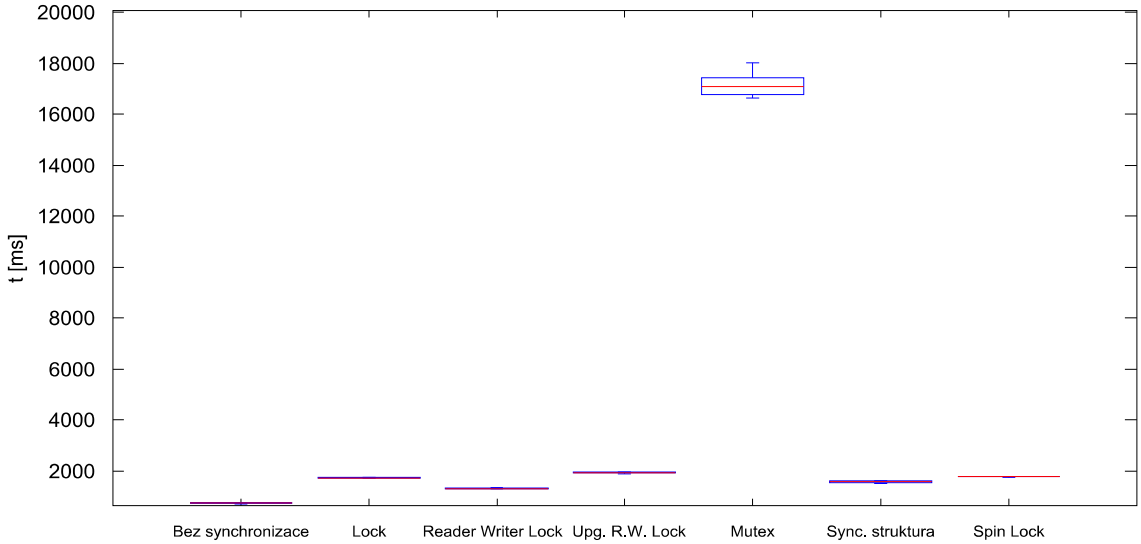
V poslední ukázce (viz. Výpis 6.5) je znázorněn zápis dat do sdíleného slovníku. Opět i tento test je prováděn hlavně kvůli metodám, které umožňují zvlášť synchronizovat čtení a zápis. Bez těchto metod by nebylo nutné provádět tři různé testy, poněvadž ostatní synchronizační metody by měly vykazovat stejnou výkonnost ve všech případech.

6.2 Výsledky testů synchronizace sdílených dat

Každý z výše uvedených testů jsem provedl 100x pro každou synchronizační metodu. Jedná se o dostatečný počet hodnot, ze kterých je již možné určit vlastnosti jednotlivých testovaných metod. Pro zobrazení jsem zvolil krabicové diagramy, vykreslené v aplikace Octave.

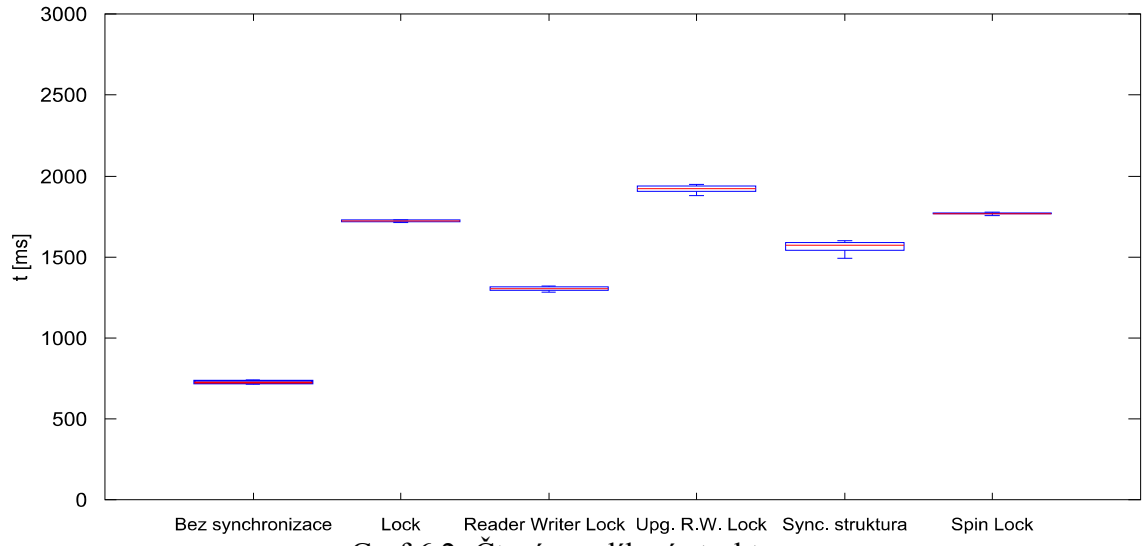
Už první test (Graf 6.1) poukazuje na značné problémy metody Mutex při synchronizaci v mých testech. Doba, za kterou byl schopný vykonat zadanou činnost, je 9-10 krát vyšší než v případě ostatních metod. Výsledek je to částečně očekávaný, jelikož Mutex není přímo součástí rozhraní .NET, ale WIN32 a jeho účelem je synchronizace na jiné úrovni. Mutexy jsou viditelné v celém systému a možností jejich využití je například meziprocesová synchronizace. Z těchto důvodů jsem se rozhodl

výsledky Mutexu v dalších diagramech neuvádět.



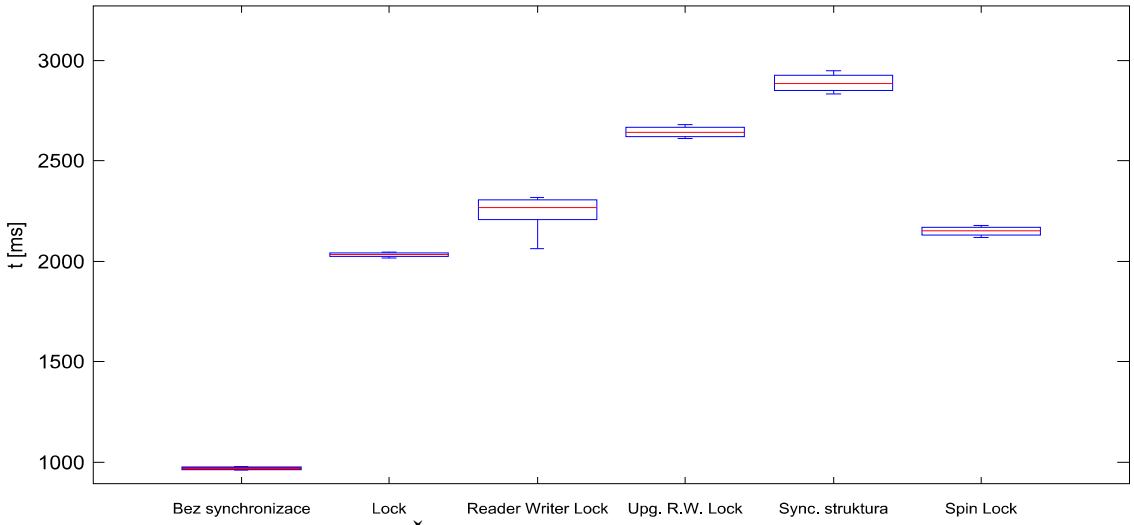
Graf 6.1: Čtení ze sdílené struktury

Graf 6.2 zobrazuje stále stejný test, pouze v čitelnější podobě bez výsledků metody Mutex. Z tohoto měření nejlépe vychází metoda Reader Writer lock, která v tomto testu využívá svojí část s názvem ReadLock, která umožňuje více čtenářům přístup ke společným datům. Čím více budou data využívána ke čtení, tím hůře se budou chovat ostatní metody oproti Reader Writer lock. Dobrého výsledku dosahuje také synchronizovaná struktura typu slovník. Tato struktura nejspíše vnitřně využívá obdobných vlastností, jakými disponuje Reader Writer lock. Téměř vyrovnané jsou si metody Lock a Spin Lock, obě metody pracují na stejném principu pouze s rozdílem, že SpinLock neuspí vlákno. Dle diagramu je na tom nejhůře upgradable read lock, který je speciální funkcí ze skupiny Reader Writer lock.



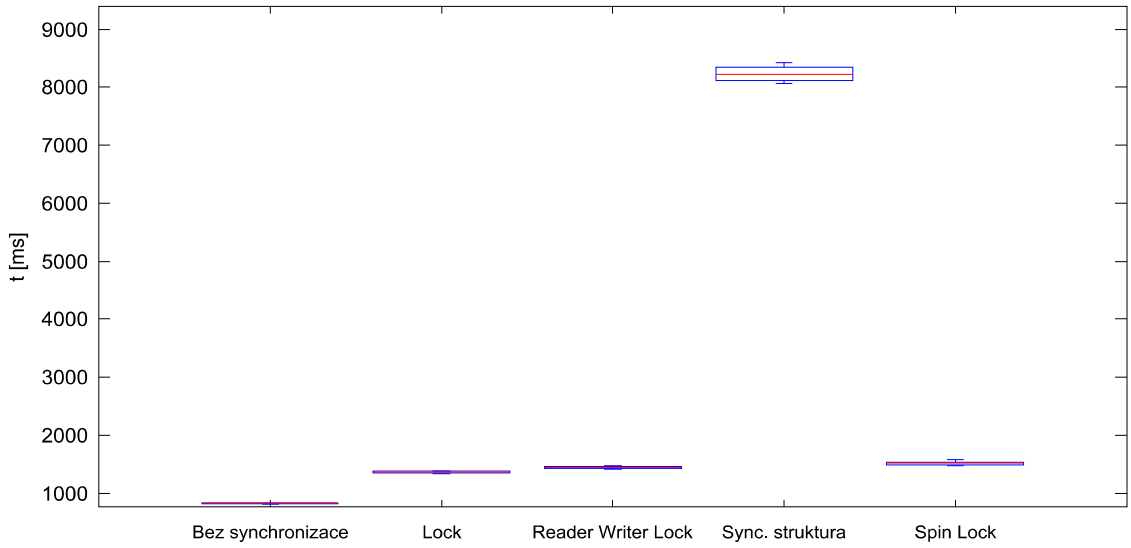
Graf 6.2: Čtení ze sdílené struktury

Na následujícím grafu (viz. Graf 6.3) jsou uvedeny výsledky testu, který se skládal ze čtení a občasného zápisu (viz. Výpis 6.4). Nejhorší výsledky zde vykazuje synchronizovaná datová struktura. Vzhledem k tomu, že jedinou změnou oproti předchozímu příkladu je příležitostný zápis, pravděpodobně v této struktuře není dostatečně optimalizován synchronizovaný zápis z více vláken současně. Opět dosti podobné výsledky vykazují metody Spin Lock a standardní Lock. V tomto testu dokonce předčily výkonnost nové metody Reader Writer lock, která však nezaostává o více než 300ms.



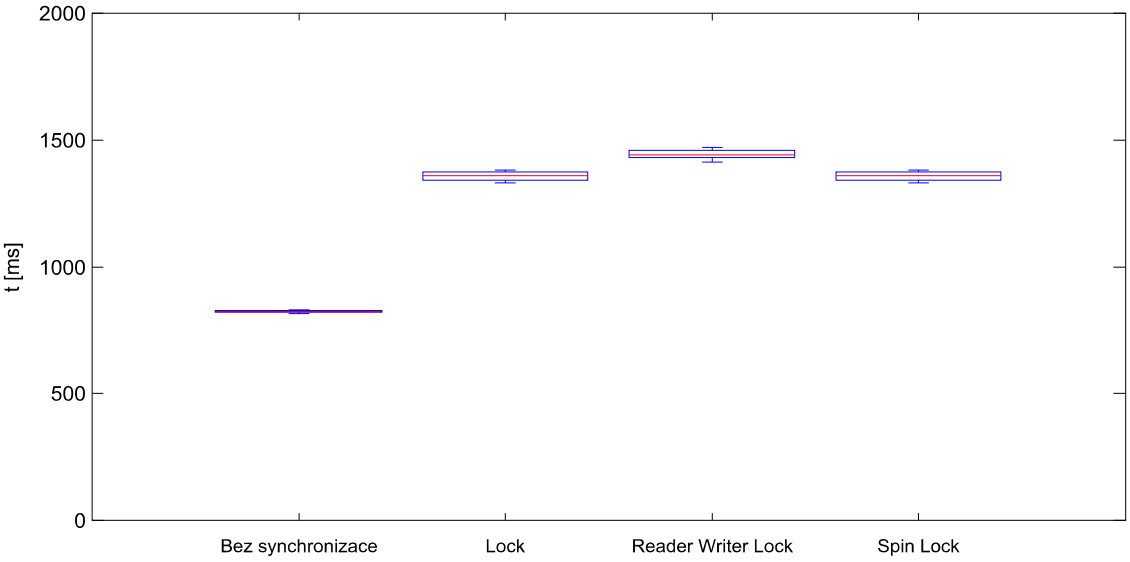
Graf 6.3:Čtení a zápis do sdílené struktury

Úkolem posledního měření bylo vyhodnocení synchronizačních schopností při zápisu do sdíleného objektu (viz. Výpis 6.5). Výsledky jsou zobrazeny v následujících diagramech (viz. Graf 6.4, Graf 6.5). V prvním z uvedených je vidět jasný velký propad synchronizované struktury oproti ostatním synchronizačním metodám. Tento propad je způsoben tím, že nová hodnota je v každé iteraci do tohoto objektu zapsána několikrát ve smyčce. V případě ostatních metod je efektivní zahrnout celou smyčku do uzamčené oblasti a vstupovat tak do zamčené oblasti pouze jednou v každé iteraci. Kdežto v případě synchronizované struktury je zamykán každý zápis zvlášť, což způsobuje výkonnostní propad, vzhledem k určité náročnosti oblast při vstupu zamknout a opět uvolnit po ukončení operace. Z výše uvedených důvodů je tedy i synchronizovaná struktura pro mé účely nevhodná.



Graf 6.4: Zápis do sdílené struktury

V dalším grafu (Graf 6.5) jsou zobrazeny výsledky ze stejného testu, pouze s rozdílem, že zde nejsou, pro vyšší čitelnost diagramu, vyneseny výsledky synchronizované struktury. Výsledky zbylých vyobrazených metod jsou v testu zápisu hodnot do synchronizovaného objektu téměř shodné.



Graf 6.5: Zápis do sdílené struktury

S posledním testem zbyly pouze tři možné způsoby synchronizace. Metoda SpinLock, která ve všech testech vykazovala obdobné výsledky jako standardní Lock metoda, není pro mé řešení nejvhodnější. Jelikož při čekání na odemčení synchronizované oblasti pouze čeká, ale neuspí jí příslušné vlákno, dosahuje dobrých výsledků pouze v případech, kdy je čekací doba malá. Avšak při delším čekání zbytečně zatěžuje systém tím, že nepředá výpočetní schopnosti dalším pracujícím vláknům. V mé aplikaci nelze předem říci, jak dlouho bude oblast zamčena, proto by mohla v některých

případech působit tato metoda problémy. Zbylé dvě metody jsou v konečném vyhodnocení všech měření do jisté míry dosti podobné. Metoda Reader Writer lock vykazuje značně lepší chování při čtení ze sdíleného objektu než standardní metoda Lock. V případě zápisu má zase mírný náskok metoda Lock. Na základě výsledků testů a také ze znalosti struktury své aplikace jsem se rozhodl použít novější metodu Reader Writer lock, která je dostupná nově v rozhraní .NET 4. Učinil jsem tak z několika důvodů. Čtení dat, ve kterém tato metoda vyniká, probíhá v mé systémové službě mnohem častěji než zápis, hlavně v případech, kdy je uživatel prostřednictvím grafické aplikace informován o stavu systémové služby a o průběhu veškerých operací. Metoda je sice náročnější z hlediska programování oproti starší běžné metodě Lock, ale na druhou stranu se jedná o sofistikovanější systém. Programátor si může dle konkrétního případu zvolit ze tří dostupných součástí, kterými jsou ReadLock, WriteLock a UpgradableReadLock. Vhodnou kombinací těchto prvků je možné vytvořit velice dobrý systém pro synchronizaci sdílených dat.

6.3 Analýza dostupných metod umožňujících paralelizaci aplikace

Po výběru vhodné metody pro synchronizaci sdílených objektů jsem se zaměřil na výběr vhodných prvků umožnujících paralelizaci vykonávané činnosti v aplikaci. Pojmem paralelizace mám na mysli rozdělení určitých vybraných činností aplikace do zvláštních vláken. Každá taková paralelní, nebo-li vícevláknová aplikace pracuje minimálně se dvěma vlákny. Vlákna jsou tedy základní stavební jednotkou takových aplikací. Zmíněná vlákna je možné v .NET vytvořit a spustit mnoha různými způsoby. Za cíl jsem si dal z dostupných způsobů vybrat ten nejvhodnější možný pro potřeby mé systémové služby. Za tímto účelem jsem vytvořil aplikaci pro otestování jednotlivých metod. Účelem testů bylo vyhodnotit vhodnost použití jednotlivých způsobů paralelizace. V testech jsem sledoval několik nejdůležitějších parametrů. Zmíněnými parametry jsou hlavně doba potřebná k vytvoření nových vláken a následně také k jejich ukončení, dále jsem sledoval potřebu paměti s ohledem na počet vláken a nakonec také využití výkonu procesoru. Vedlejším faktorem pak jsou navíc způsoby implementace a nabízené funkce dostupné při programování. Vybranými metodami jsou jak metody dostupné již ve starších verzích .NET, tak i metody nově uvedené v .NET 4.

K měření potřebných hodnot jsem použil převážně systémové výkonnostní čítače, které měli za úkol sledovat stav vybraných parametrů. Prvním z parametrů byl

počet existujících vláken v testovací aplikaci. Dále jsem monitoroval velikost soukromé paměti alokované aplikací. Nakonec také procentuální využití času procesoru programem. Přímo v testovacím programu jsem navíc měřil, s použitím metody *StopWatch* dobu, za kterou všechna zvolená vlákna vykonají zadanou činnost.

Hlavním cílem testu nebylo testování absolutní výkonnosti při vykonávání náročných výpočetních operací. Cílem testu bylo prověřit schopnosti systému vytvářet vlákna, rušit je a pracovat s nimi při použití jednotlivých vybraných metod paralelizace. Testovací postup se skládal ze současného spuštění předem nastaveného počtu vláken. Každé vlákno mělo za úkol po svém spuštění inkrementovat proměnnou uchovávající počet spuštěných vláken, po té bylo jejich úkolem počkat na spuštění zbylých vláken. Ve chvíli, kdy byla všechna požadovaná vlákna spuštěna, každé vlákno čekalo 10 s. Po uplynutí požadované doby deseti sekund byla požadovaná činnost vláknem vykonána a vlákno se tedy mohlo ukončit. Na následujícím výpisu (Výpis 6.6) je zobrazena část kódu, kterému se jednotlivá vlákna během své činnosti věnují.

```
private static void Work()
{
    lock (zamek)
    {
        total++;  //Signalizace spusteni vlakna
    }
    while (total < cnt)  //cekani na spusteni zbylych vlaken
    {
        Thread.Sleep(10);
    }
    Thread.Sleep(10000);    //pauza 10s
}
```

Výpis 6.6: Testovací funkce

Při testování jsem jako výchozí počet vláken zvolil 100. Tento počet považuji za dostatečný vzhledem k potřebám mé systémové služby. V tomto množství by současně mělo být možné odhalit případné nedostatky a problémy jednotlivých metod použitých k paralelizaci.

Testu jsem podrobil několik následujících metod. Tou nejznámější je metoda *Thread*, která se nachází v systémové knihovně *System.Threading*. Při použití této metody je vytvořeno přímo nové vlákno, jak již napovídá i název Thread = Vlákno. Další možností pro vytváření vláken, kterou jsem podrobil testům, je přímé použití *ThreadPool*. I tato metoda je ze systémové knihovny *System.Threading*. V případě této metody vytváří nová vlákna systém a ne přímo programátor, ten pouze zadá

požadovanou činnost a vlákna jsou založena dle potřeby. Pro paralelizaci jsem také otestoval možnost použití asynchronního delegáta. Delegáty se chovají obdobně jako použití metody z *ThreadPool*. Programátor pouze zadá požadovanou činnost a systém zařídí její vykonání. Další metodou, kterou jsem podrobil testům je *BackgroundWorker* ze systémové knihovny *System.ComponentModel*. Tato metoda se používá, díky některým svým vlastnostem, převážně v aplikacích s grafickým prostředím, ale nic nebrání jejímu použití v systémové službě. Poslední dvě testované metody jsou dostupné od nové verze .NET a nachází se taktéž v systémové knihovně, tentokrát však *System.Threading.Tasks*. První z nich je metoda *Task,* způsob použití se do jisté míry podobá metodě *Thread*, ale z funkčního hlediska jsou dosti odlišné. Poslední z testovaných metod je paralelní metoda *for* s názvem *Parallel.For.* Pracuje obdobným způsobem jako klasické, všem známé, sekvenční *for,* avšak jednotlivé iterace nejsou prováděny postupně ale současně.

6.4 Výsledky testu paralelizačních metod

ThreadPool

ThreadPool existuje již od první verze .NET. V systému spravuje určitý počet vláken, které nabízí k dispozici pro asynchronní úkoly. V prvních verzích .NET měl ThreadPool omezen maximální počet spravovaných vláken na přibližně 25. V takovém případě mohl systém současně vykonávat pouze 25 úkolů a ostatní úkoly ve frontě musely čekat. V současných verzích .NET je velikost ThreadPool dána parametry systému a nabízí řádově až 1000 vláken, což je pro potřeby mé aplikace více než dostatečné množství. Jednotlivé úkoly jsou zařazovány do fronty, kterou ThreadPool sleduje a postupně jim přiřazuje vlákna, ve kterých se požadovaná činnost vykoná. Způsob jakým jsem ThreadPool pro paralelizaci použil, je znázorněn v následujícím výpise (viz. Výpis 6.7). Zjevnou výhodou oproti použití metody Thread, která vytváří přímo nová vlákna, je jednoduchost použití. Programátor zde pouze předá úkol, který by se měl vykonat v novém vlákně, frontě a dále se o jeho vykonání stará systém.

```
WaitCallback clb = new WaitCallback(obj => { Work(); });
for (int i = 0; i < 100; i++)
{
    ThreadPool.QueueUserWorkItem(clb);
}
```

Výpis 6.7: Způsob použití ThreadPool

Jak jsem již dříve zmínil, v tomto testu jsem sledoval dobu, za kterou dojde ke spuštění všech požadovaných vláken. Dále jsem také vyhodnocoval dobu, za kterou jsou vlákna ukončena a odebrána ze systému po vykonání požadovaného úkolu. Na grafu (Graf 6.7) je znázorněn průběh spouštění a následné ukončení vláken v testovací aplikaci. Doba, po kterou trvalo spuštění všech 100 vláken činila v průměru 98s. Čas, kdy všechna vlákna vykonala požadovanou činnost dosahuje hodnoty 10s od spuštění všech vláken. V ideálním případě by mělo dojít k postupnému ukončení všech vláken po dokončení veškeré činnosti, tedy po uplynutí 10s od spuštění všech vláken. U této metody jsou však vlákna řízena automaticky systémem a k jejich korektnímu ukončení dojde až tehdy, kdy k tomu systém vydá pokyn. V tomto případě dojde k ukončení po 18s od vykonání zadané činnosti. Celkový čas testu tedy činí 126s.

Asynchronní delegát

Další testovanou možností je asynchronní použití delegáta. Delegáty fungují v .NET jako odkazy na funkce nebo metody. Obdobně jako v jiných jazycích existují ukazatele na funkce i v C#. Tuto funkci umí zastat delegát, který tvoří objekt odkazující na zadanou funkci. Delegáty umožňují jak synchronní spuštění metody, na kterou odkazují tak i asynchronní. V případě synchronního je funkce zpracována ve vlákně, ze kterého ji delegát spustil. U asynchronního dojde ke spuštění nového vlákna, ve kterém proběhne zvolená funkce nebo metoda. Způsob, jakým jsem metodu použil, je znázorněn na následujícím výpise kódu (viz. Výpis 6.8). Podmínky testu jsou stejné jako v předchozím. Úkolem bylo spuštění 100 vláken. Paralelizaci zde zajišťuje zavolání funkce BeginInvoke, která přebírá dva parametry. První z nich je typu *AsyncCallback*, který odkazuje na metodu, která má být zavolána po dokončení funkce, která je delegátovi zadána. Druhý parametr je typu *Object* a může jím být libovolný objekt, který je nutné předat zmíněné callback metodě. V mém případě však nebylo nutné ani jeden z dostupných parametrů použít.

```
AsyncDelegat Delegat = new AsyncDelegat(() => { Work(); });
for (int i = 0; i < 100; i++)
{
    Delegat.BeginInvoke(null, null);
}
```

Výpis 6.8: Asynchronní použití delegáta

Výsledky testu, v tomto případě průběh spouštění jednotlivých vláken, jsou opět

znázorněny v grafu (viz.Graf 6.7). Už na první pohled je z průběhu spouštění vláken zřejmé, že je dosti podobný s předchozím testem, ve kterém byl testován ThreadPool. V tomto případě trvalo spuštění všech požadovaných vláken také v průměru 98s. Doba, která uplynula od ukončení veškeré činnosti k fyzickému ukončení vláken činí 18s.

Z naměřených výsledků je zřejmé, že na pozadí obou doposud testovaných metod pracuje pravděpodobně stejný systém, který se stará o vytváření vláken.

Paralelní For

Tato metoda je z nové verze .NET. Hlavním důvodem jejího vzniku je pravděpodobně usnadnění paralelizace náročných výpočetních nebo jakýchkoliv jiných iteračních metod. Způsob jejího použití je v principu stejný jako v případě běžného for. Zadávají se meze odkud kam má iterace proběhnout a také metoda, která má být spuštěna, není však možné nastavit krok. Metoda navíc nabízí určité možnosti nastavení paralelizace. Tyto možnosti jsem však v mém případě nevyužil, jelikož tato nastavení umožňují hlavně nastavit úroveň paralelizace zadané činnosti. V mém případě však potřebuji pro každou zadanou činnost jedno vlákno a tak nastavení úrovně paralelizace postrádá smysl. Způsob mého použití je znázorněn v následujícím výpise (viz. Výpis 6.9).

```
Parallel.For(0, 100, a => { Work(); });
```

Výpis 6.9: Paralelní for

Na grafu níže (viz. Graf 6.7) je opět znázorněn průběh spouštění vláken během testu. Průběh opět dosti podobný předchozím dvěma metodám, tentokrát však byla doba spouštění vláken o 2s kratší a činila 96s. Doba, která uplynula od ukončení činnosti do fyzického ukončení vláken byla opět 18s. Průběh viditelný v grafu opět nasvědčuje tomu, že k zakládání nových vláken je použit stejný systém jako u předchozích metod. Rozhodl jsem se tedy najít důvod z jakého je tato metoda o 2s rychlejší než předchozí i přes to, že používá obdobný systém správy nových vláken. Při bližším pohledu na naměřená data jsem objevil, že tato metoda využívá při testu o jedno vlákno méně než ostatní testované metody. A zjistil jsem tedy, že důvod proč tato metoda ušetří spouštění jednoho vlákna je způsoben tím, že využije i vlákno, ze kterého byla spuštěna. Blokuje tak tedy i hlavní vlákno, které jí spustilo, což může z hlediska

mé aplikace v některých případech způsobit problém.

Task

Tato metoda také patří mezi nově uvedené. Nabízí jí nejnovější verze .NET. Práce s tasky je obdobná jako s thready. Tasku se prostřednictvím delegáta nebo lambda výrazu zadá funkce nebo úkol, který má být vykonán a následně je možné task spustit zavoláním metody Start. V případě tasku zajišťuje spouštění a ukončování vláken nová knihovna nesoucí název Task Parallel Library (TPL). Programátor tasku zadá úkol, který je nutné vykonat a TPL zajistí jeho vykonání. Nemusí tak vždy platit, že jeden task zaujme v systému jedno vlákno, na základě vyhodnocení TPL může například více tasků sdílet jedno vlákno a podobně. V následujícím výpise (viz. Výpis 6.10) je znázorněn způsob použití tasku v mém testu. Došlo ke spuštění 100 tasků, kde každý dostal prostřednictvím lambda výrazu úkol. Úkolem byla opět již několikrát zmíněná testovací funkce (viz. Výpis 6.6).

```
for (int i = 0; i < 100; i++)
{
    new Task(() => { Work(); }).Start();
}
```

Výpis 6.10: Task

Přestože se řízení této metody opírá o novou knihovnu Task Parallel Library, z naměřených dat a průběhu zobrazujícího spouštění nových vláken (viz. Graf 6.7) je zřejmé, že jsou vlákna vytvářena podle stejného řádu jako v předchozích metodách. Z toho důvodu jsou i naměřené výsledky této metody dosti podobné s předchozími metodami. Spuštění všech požadovaných vláken trvalo v průměru 96s, je tedy přibližně o 2s rychlejší něž asynchronní použití delegáta nebo metody ThreadPool. K ukončení všech vláken došlo po 18s od ukončení jejich činnosti, což je totožné s naměřenými hodnotami všech předchozích testů.

Background Worker

Jednou jsem již zmínil, že background worker vznikl hlavně pro potřeby grafických aplikací. Jeho úkolem je vykonávat časově náročné úkoly v odděleném vlákně a neblokovat tak při práci uživatelské rozhraní. Není však žádný problém v použití background workeru bez grafického uživatelského rozhraní. Princip background workeru je obdobný jako u jiných metod. Programátor mu zadá

prostřednictvím nové události požadovanou činnost a následně ho může kdykoliv zavoláním příslušné metody spustit. Backgroundworker po spuštění zajistí vytvoření nového vlákna. Zajímavostí této metody je například schopnost reportovat postup činnosti prostřednictvím nastavených událostí. Dále také nabízí jednoduše použitelné metody pro zrušení nebo pozastavení činnosti. Ve výpise níže (viz. Výpis 6.11) je znázorněn způsob, jakým jsem použil tuto metodu v testu. Vytvořil jsem pole backgroundworkerů, kterým jsem v cyklu přiřadil požadovanou činnost. Po přiřazení úkolu jsem metodou RunWorkerAsync spustil jejich prostřednictvím vykonávání zadaného úkolu.

```
BackgroundWorker[] bws = new BackgroundWorker[100];
for (int i = 0; i < 100; i++)
{
    bws[i] = new BackgroundWorker();
    bws[i].DoWork += new DoWorkEventHandler(WorkBW);
    bws[i].RunWorkerAsync();
}
```

Výpis 6.11: Background worker

Výsledný průběh spouštění nových vláken pří použití této metody je vynesen v grafu (viz. Graf 6.7). I tato metoda, obdobně jako ostatní doposud testované metody, využívá k vytváření nových vláken stejný systém. Vytvoření jednoho nového vlákna trvá přibližně 1s. Celkový čas pro spuštění všech požadovaných vláken činil v průměru 97s. K ukončení zadané činnosti došlo opět po stejné době od spuštění všech vláken. Od tohoto času jsem měřil dobu, za jakou dojde k odebrání vytvořených vláken ze systému. I v tomto případě se doba, za kterou byla vlákna odebrána ze systému, rovnala 18s. I tento fakt nasvědčuje stejnému základu pro správu nových vláken v systému jako u předchozích testovaných metod.

Thread

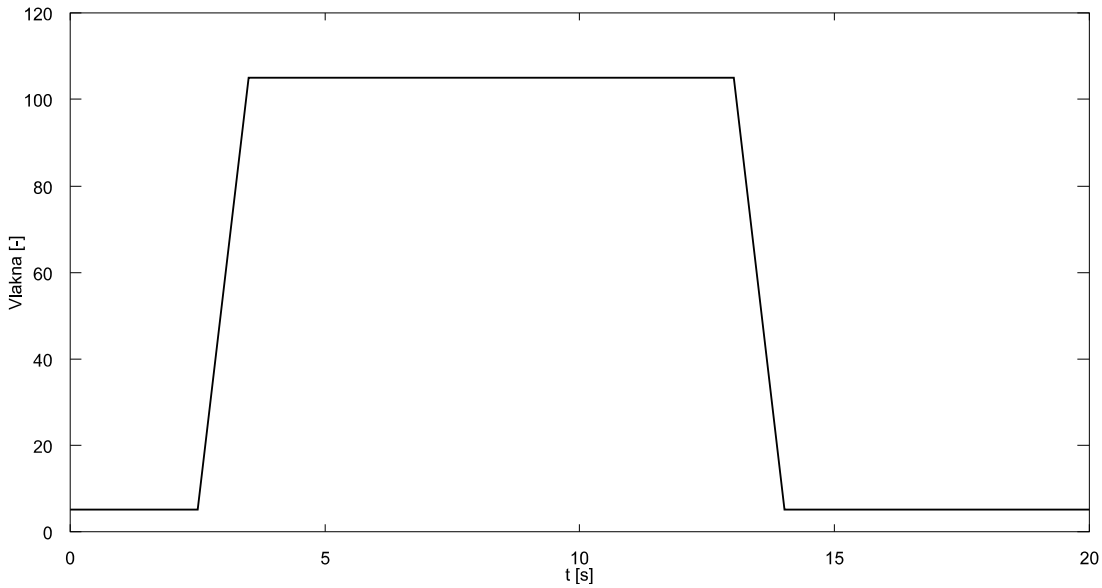
Poslední metodou, kterou jsem podrobil testům je metoda Thread. Jde o nejzákladnější metodu, na kterou se dá narazit při práci s vícevláknovými aplikacemi. Tato metoda se neopírá o žádnou vnitřní systémovou správu pro vytváření nových vláken. V tomto případě je zodpovědnost při vytváření vláken přímo v rukou programátora. Při použití metody thread jsou vlákna vytvářeny přímo. Jako jediná z testovaných metod umožňuje vytvářet vlákna, která budou pracovat, jak v popředí, tak i na pozadí. V případě ostatních metod pracovala vlákna vždy na pozadí. Vlastností

vláken pracujících v popředí je, že k jejich ukončení nedojde, dokud nevykonají zadanou činnost ani v případě, že dojde k ukončení hlavního vlákna, ze kterého byly vytvořeny. Způsob, jakým jsem vytvářel vlákna metodou Thread v tomto testu, je znázorněn v následujícím výpise (viz. Výpis 6.12). Je vytvořen nový objekt typu Thread, jemu je předána instance metody, která má být v novém vlákně vykonána. Zavoláním metody Start je nové vlákno v systému vytvořeno a spuštěno.

```
for (int i = 0; i < 100; i++)
{
    new Thread(Work).Start();
}
```

Výpis 6.12: Thread

Vzhledem k tomu, že je možné odečítat hodnoty systémových výkonnostních čítačů nejrychleji přibližně každých 500ms je výsledný graf tohoto testu (viz. Graf 6.6) zkreslený s ohledem na rychlé změny sledovaného počtu vláken. Dle naměřených hodnot bylo spuštěno 100 vláken v průměru za 33ms. Tento výsledek je několikanásobně lepší než hodnoty dosažené kteroukoliv jinou testovanou metodou. Tento výkonnostní skok je způsoben hlavně tím, že k vytváření vláken dochází přímo na příkaz programátora a ne prostřednictvím systémového správce vláken. Ukončení vláken je také okamžité. Po dokončení zadané činnosti se každé vlákno samo ukončí a odebere ze systému.

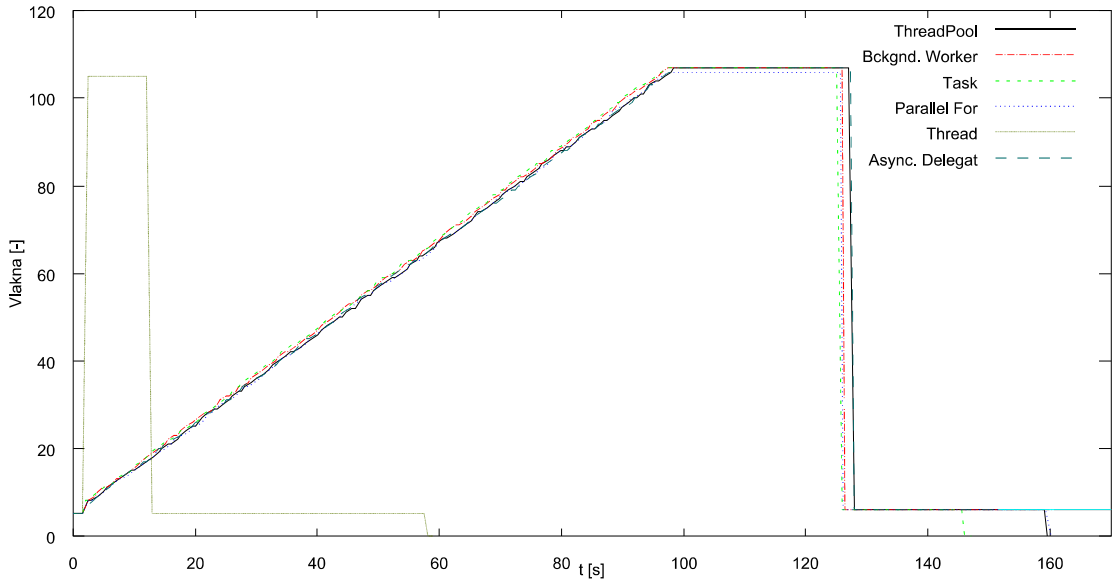


Graf 6.6: Thread

Vzhledem k potřebám mé aplikace se pro cyklické operace podle dosavadních

testů nejvíce hodí poslední testovaná metoda nesoucí název Thread. Tato metoda vyniká hlavně svou rychlostí. Nenabízí sice žádné doplňkové funkce, které obsahují ostatní metody, jako například reportování průběhu zadané činnosti nebo předávání vzniklých výjimek z nového vlákna zpět do hlavního. Nejedná se ani o nejjednodušší metodu s ohledem na způsob použití, ale programátor má v jejím případě přímou kontrolu nad vytvářenými vlákny.

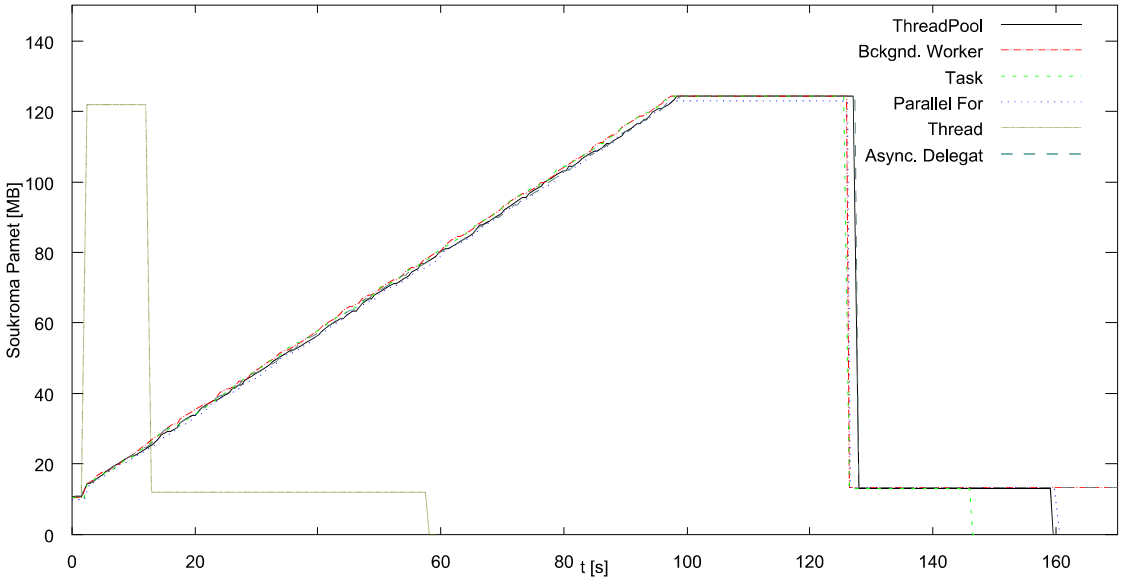
Na následujícím grafu (viz. Graf 6.7) jsou zaneseny výsledky všech testovaných metod. Tento graf potvrzuje závěry, které jsem shrnul výše. Metoda, která je schopná vykonat zadanou činnost nejrychleji se nazývá Thread. Ostatní metody, jak jsem již zmínil, dosahují velice podobných výsledků a je tedy pravděpodobné, že využívají stejný systém správy vláken. U metody thread jsou spouštěna vlákna přímo na pokyn programátora a vykazuje tedy podstatně lepší výsledky v této úloze.



Graf 6.7: Všechny testované metody

Další veličinou, kterou jsem v tomto testu pro jednotlivé metody sledoval bylo využití soukromé paměti. Vzhledem k výsledkům, které jsem při tomto testu naměřil jsem se rozhodl umístit výsledné průběhy soukromé paměti pro jednotlivé metody do jednoho grafu (viz.Graf 6.8). Z naměřeného průběhu je na první pohled patrná podoba s předchozím grafem (Graf 6.7). Jelikož v tomto testu testuji pouze schopnosti metod vytvářet a ukončovat nová vlákna, alokovaná paměť se mění pouze s počtem vláken. Z naměřených hodnot je nejúspornější metodou metoda thread, při jejímž testu testovací

aplikace využívala 122MB soukromé paměti. Tato metoda je nejúspornější vzhledem k paměti díky své jednoduchosti, nenabízí žádně složité funkce. Druhou nejúspornější je metoda využívající paralelní for. Tato metoda alokuje v testu průměrně 123MB. To je přibližně o 1MB méně než zbylé metody. Tato úspora vznikla tím, že metoda paralelní for využívá pro činnost i hlavní vlákno a potřebuje tak v součtu o jedno vlákno méně než ostatní. Zbylé metody jsou si co do využití paměti rovné a využívají tak 124MB soukromé paměti. Všechny hodnoty jsou si velice blízké a 2MB operační paměti rozdílu na 100 vytvořených vláken jsou v dnešní době téměř zanedbatelné. Všechny metody tedy co do alokace soukromé paměti hodnotím jako srovnatelné.



Graf 6.8: Využití soukromé paměti

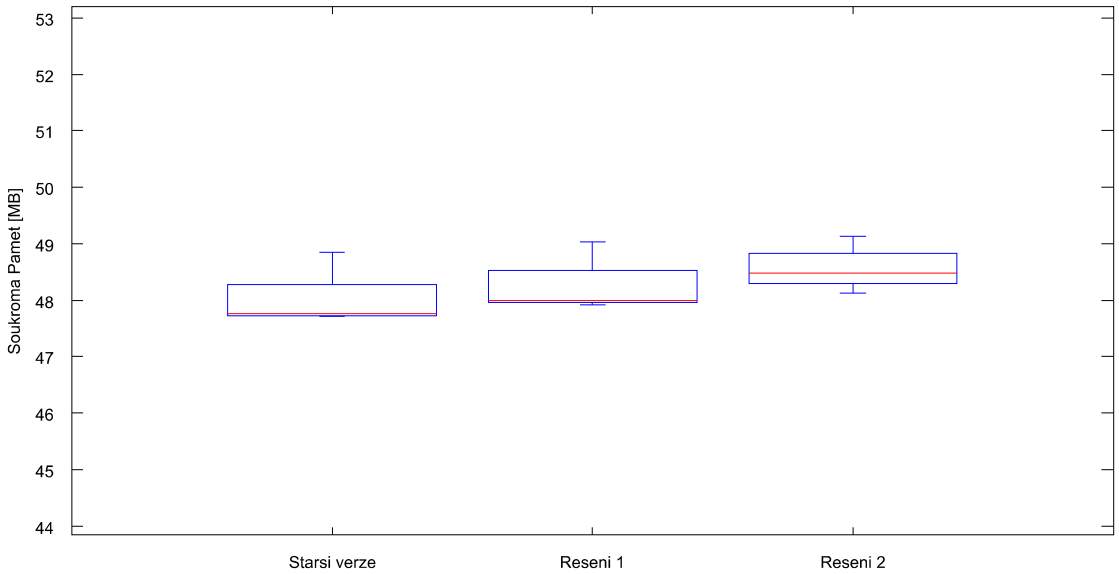
Při testech jsem sledoval také průběžné využití výpočetního výkonu procesoru. Výsledky naměřených hodnot však ukázaly, že ani jedna z testovaných metod nezatěžuje žádným významným způsobem procesor. Využití procesoru jsem měřil prostřednictvím výkonnostních čítačů, u kterých je tento parametr vyjádřen v procentech. V průběhu měření tato hodnota nedosahovala ani jednoho procenta. Největší vytížení bylo pozorovatelné u metody thread, která je schopná vytvořit požadovaný počet vláken za velmi krátký časový okamžik, u ostatních metod je toto vytížení rozloženo do doby, po kterou se spouští vlákna. Ve vytížení procesoru si tedy metody nejsou úplně rovny, ale hodnoty vytížení jsou i pro relativně velký počet vláken tak malé, že není nutné se jimi výrazně zabývat. Z výsledků všech testů je nejvhodnější metodou pro paralelizaci cyklických operací metoda Thread.

7 Dosažené výsledky

7.1 Využití paměti

Na následujících řádcích jsem zhodnotil výsledky, kterých dosahují jednotlivá řešení mé systémové služby. Porovnával jsem schopnosti neoptimalizované, mnou zvané Starší verze, s dvěma novými návrhy řešení. Řešení 1 odpovídá centrálnímu správci operací a Řešení 2 pro decentralizovaný systém. V nových strukturách byly použity metody z předchozích testů, tedy metoda Thread pro paralelizaci a R-W lock pro synchronizaci. Služba byla nastavena na cyklické stahování nastavených dat z nastavených přístrojů s periodou stahování 2 minuty.

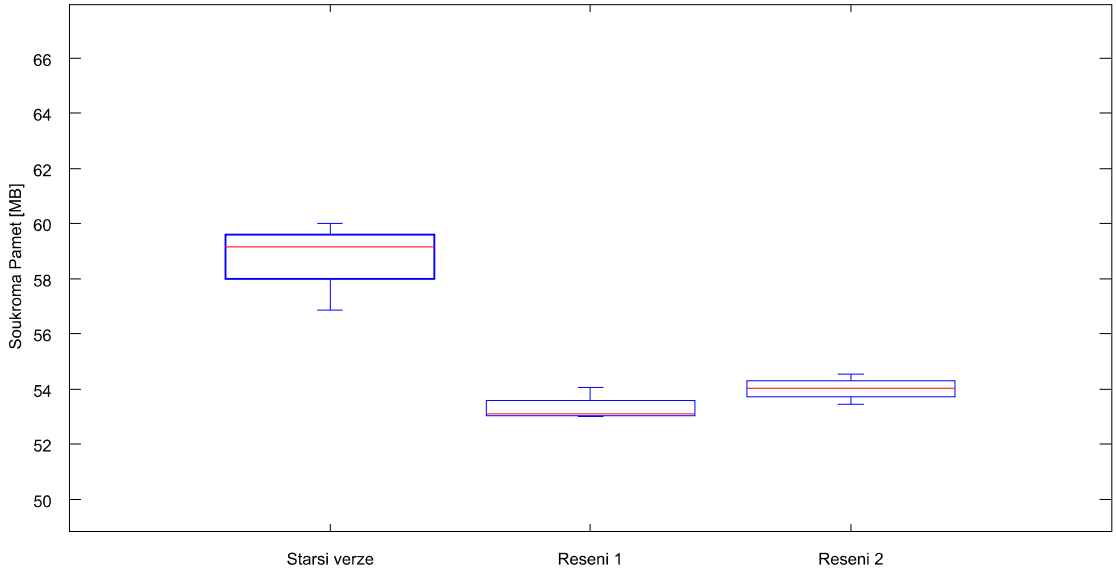
V prvním testu jsem porovnával výsledky jednotlivých řešení v klidovém stavu. Tedy ve stavu, kdy nejsou do seznamu zadány žádné přístroje a tím pádem ani operace. Výsledky tohoto testu jsou uvedeny na následujícím grafu (viz. Graf 7.1). Z grafu je zřejmé, že jednotlivá řešení jsou takřka srovnatelná. Nejlepšího výsledku dosahuje starší verze aplikace. Rozdíly jsou však tak nepatrné, že jím v tomto případě nepřikládám žádný význam. Drobný rozdíl může být způsoben změnou ve struktuře nebo například také používáním knihoven z nové verze .NET 4.



Graf 7.1: Test využití paměti – 0 přístrojů, 0 operací

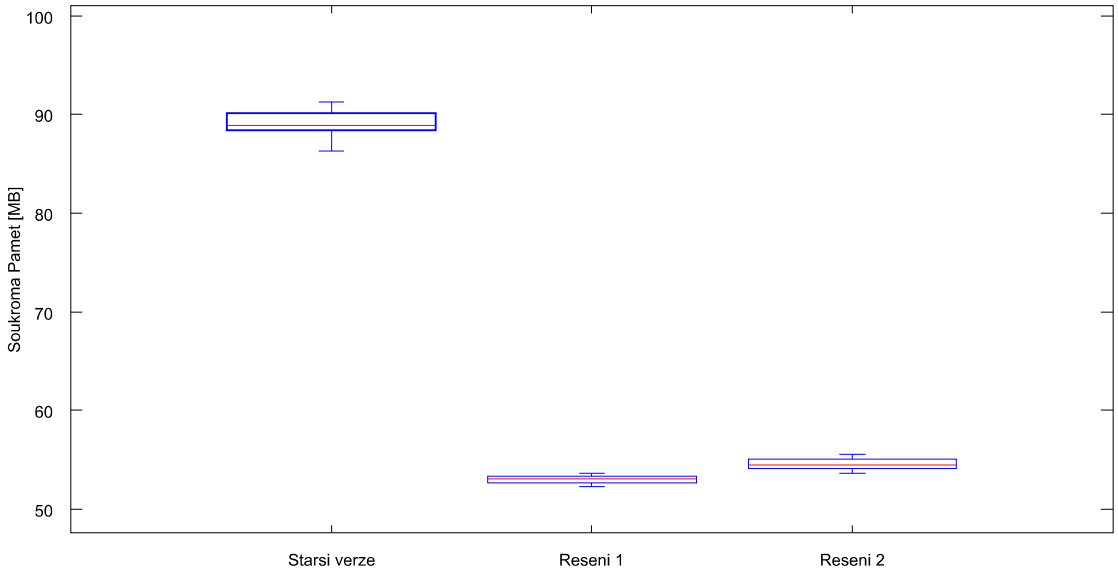
Na následujícím grafu (viz. Graf 7.2) jsou zobrazeny výsledky druhého testu. Při

testu bylo stahováno 8 různých druhů dat z jednoho měřícího přístroje. Použití jednoho přístroje s 8 operacemi považuji za velmi nízké vytížení systémové služby. Ale už i při takto nízké zátěži je patrný značný rozdíl ve využití paměti mezi starou podobou systémové služby a těmi novými, hlavně při porovnání nárůstu oproti předchozímu testu. Potřeba soukromé paměti u staré struktury narostla o více než 10MB. U zbylých dvou testovaných podob se pohybuje maximálně mezi 5 až 6MB. Dále je však vidět, že nárůst potřeby paměti u Řešení 2 je přibližně o 1MB vyšší než při Řešení 1. To je způsobeno nejspíše větším počtem použitých objektů, v tomto případě časovačů pro každou operaci, u tohoto řešení.



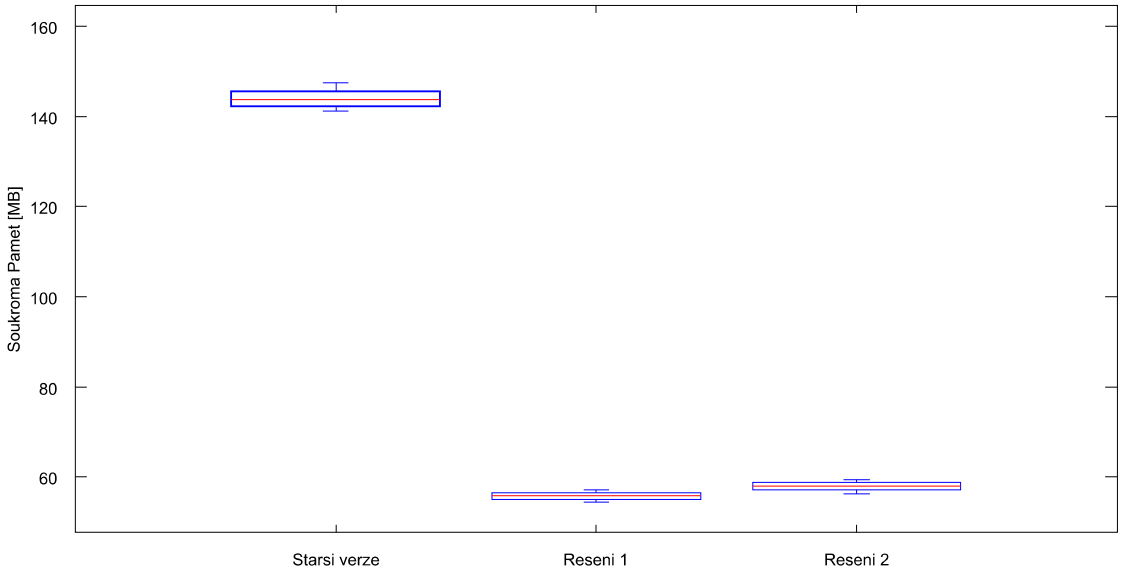
Graf 7.2: Test využití paměti - 1 přístroj, 8 operací

V dalším testu (viz. Graf 7.3) jsem již přešel k vyššímu počtu operací, pro jejich provoz musela systémová služba použít 26 vláken. Operací tedy bylo 26 a data byla stahována ze 3 přístrojů. Dva přístroje po 9 operacích a jeden s 8. Opět jsem zaznamenal vysoký nárůst využité paměti oproti předchozímu testu u starší verze. Dá se říci, že platí „pravidlo“, o kterém jsem se dříve v práci zmiňoval a to, že každé nové vlákno zabírá mezi 1-2MB paměti. V tomto případě 26 nových vláken oproti klidovému stavu se rovná nárůstu využití paměti o necelých 50MB. U dvou nových struktur nedošlo téměř k žádně změně oproti předchozímu testu. Už to považuji za velice pozitivní výsledek ve prospěch nových struktur.



Graf 7.3: Test využití paměti - 3 přístroje, 26 operací

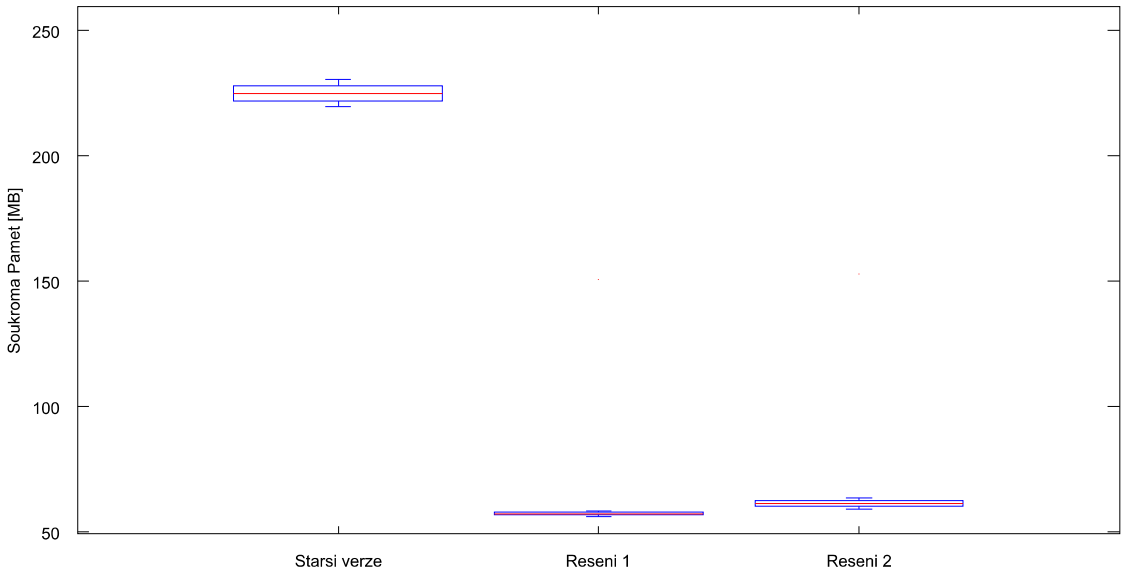
Následující test (viz. Graf 7.4) již řadím do kategorie střední zátěže. Úkolem služby bylo zpracovávat, dle stejných podmínek jako v jiných testech, 54 operací z 6 zadaných přístrojů. Tentokrát tři přístroje po 9 operacích a zbylé dva po 8. Starší verze se drží stále stejného trendu v nárůstu spotřeby paměti. Zbylá dvě řešení zaznamenala mírný nárůst v řádu 1-2MB oproti předchozímu testu. I po vyhodnocení tohoto testu stále platí, že nejšetrněji s operační pamětí zachází Řešení 1, které využívá centrální správu operací.



Graf 7.4: Test využití paměti - 6 přístrojů, 54 operací

Poslední test (viz. Graf 7.5) jsem zařadil do jakési kategorie vyšší zátěže. Opět bylo použito 6 přístrojů, tentokrát však bylo provozováno 108 operací. V tomto testu

stará verze systémové služby využívá konstantě přes 220MB, což je dosti vysoké číslo vzhledem k tomu, že ke stahování dat dochází každé 2 minuty a v mezidobí nedochází k žádné významné činnosti. Řešení 1 a 2 si i v tomto testu drží svůj standard s využitím paměti do 60MB.



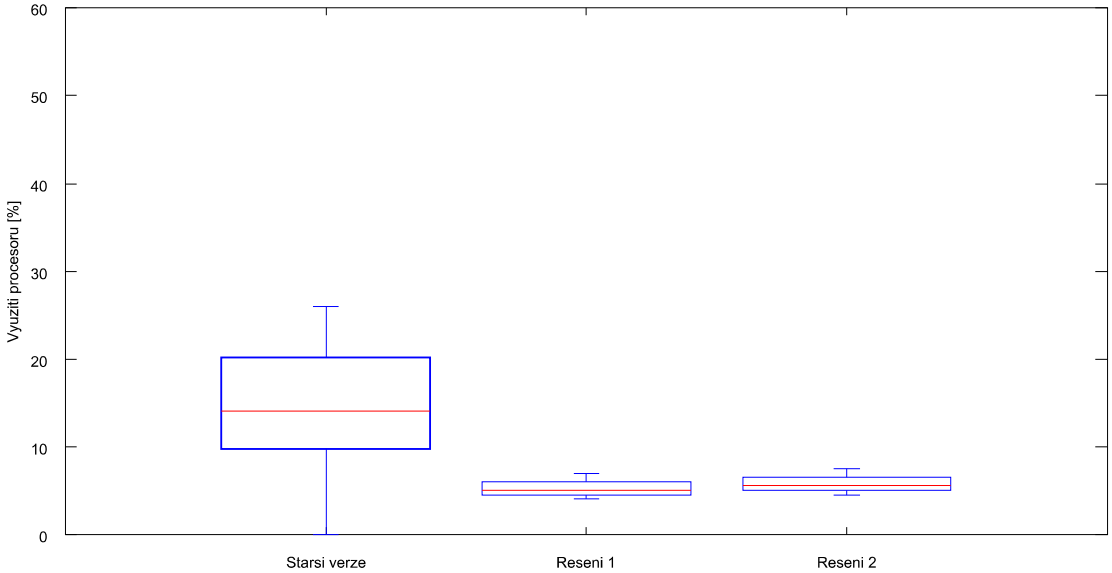
Graf 7.5: Test využití paměti - 6 přístrojů, 108 operací

Výsledky testu paměti jsou velice uspokojivé. Podařilo se mi eliminovat jeden z nejvýraznějších problémů staré struktury systémové služby, kterým byla vysoká paměťová náročnost při vykonávání zadané činnosti. Při zhodnocení dvou nových řešení prozatím nejlépe vychází varianta s centrálním správcem operací, který používá pouze jeden vylepšený časovač pro spouštění operací.

7.2 Využití procesoru

V tomto testu jsem se snažil otestovat úroveň vytížení procesoru v případě jednotlivých řešení. Vzhledem k charakteru činnosti, která je po systémové službě požadována, kdy příležitostně, není vytížení procesoru stálé a k většímu nárůstu dochází hlavně při stahování. Největšího vytížení procesoru dosahuje systémová služba logicky při největším počtu operací. Uvádím tedy přímo výsledky testu (viz. Graf 7.6) s 6 přístroji a 108 operacemi. Z výsledků je hned na první pohled patrný velký rozptyl hodnot v případě starší verze aplikace. Tento jev přikládám způsobu, kterým je řešeno načítání aktuálních konfiguračních parametrů v době, kdy nedochází ke stahování dat. V případě starší verze se každá operace starala sama o sebe a proto si musela zajišťovat i načítání zmíněných konfiguračních parametrů. Nebylo tedy možné nechat vlákna spát

po dlouhou dobu a tak i v době nečinnosti byla v určitém intervalu několika sekund vlákna probuzena za účelem načtení případných změn konfigurace. K tomuto jevu již nedochází v nových verzích, kde jsem se na eliminaci příliš častého načítání dat zaměřil. Současně s eliminací tohoto negativního jevu a výrazného snížení počtu dlouhodobě běžících vláken došlo také k výraznému snížení využití procesoru. Obě nová řešení jsou ve využití procesoru srovnatelná a lepší oproti starší verzi.



Graf 7.6: Využití procesoru - 6 přístrojů, 108 operací

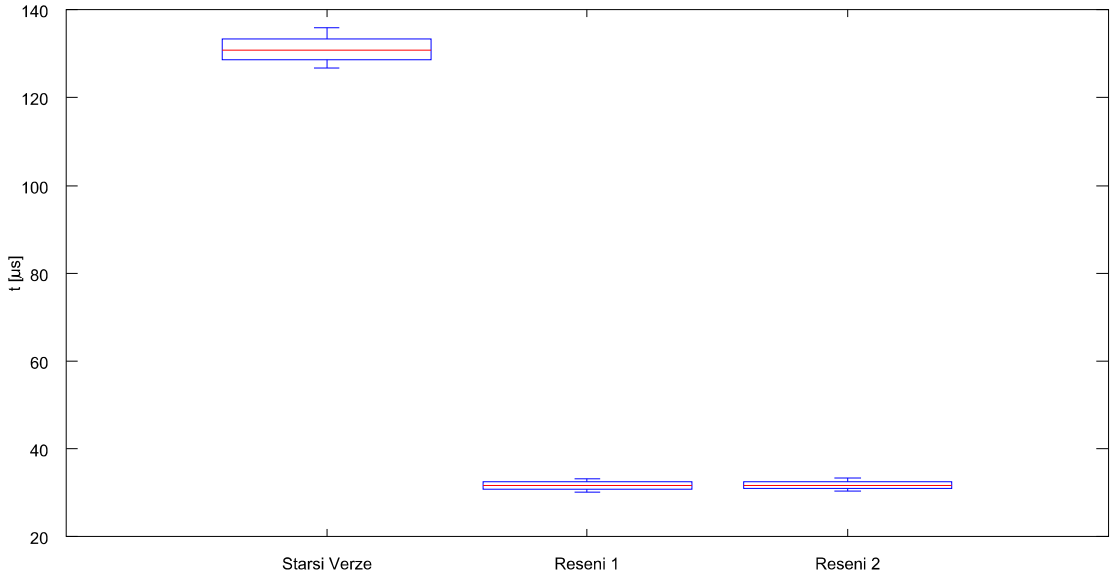
7.3 Doba potřebná pro načtení parametrů

Účelem následujícího testu bylo hlavně prověřit vhodnost použitých metod synchronizace sdílených dat v praktickém využití. Test spočíval v měření doby potřebné k načtení požadovaných dat. V prvním testu jsem měřil dobu, za kterou dojde k načtení seznamu zadaných přístrojů (viz. Graf 7.7). Druhý test (viz. Graf 7.8) probíhal obdobně, pouze s rozdílem, že docházelo k načtení seznamu zadaných cyklických operací. Oba načítané seznamy se nachází v oblasti dat, která je sdílená v rámci všech vláken v systémové službě.

Prezentované grafy jsem naměřil v testu, ve kterém systémová služba obsluhovala 6 měřících přístrojů a 108 cyklických operací. Velký počet operací a přístrojů představuje vyšší konkurenci mezi jednotlivými vlákny při přístupu do sdílené oblasti. Z toho důvodu se vylepšené způsoby nových řešení projeví nejvíce v tomto provedeném testu. Naměřené časy uvedené v následujících grafech jsou v µs. Hodnoty se tak mohou z pohledu člověka zdát jako zanedbatelné, ale z pohledu aplikace hraje i vylepšení v těchto oblastech svou roli vzhledem k četnosti jejich načítání

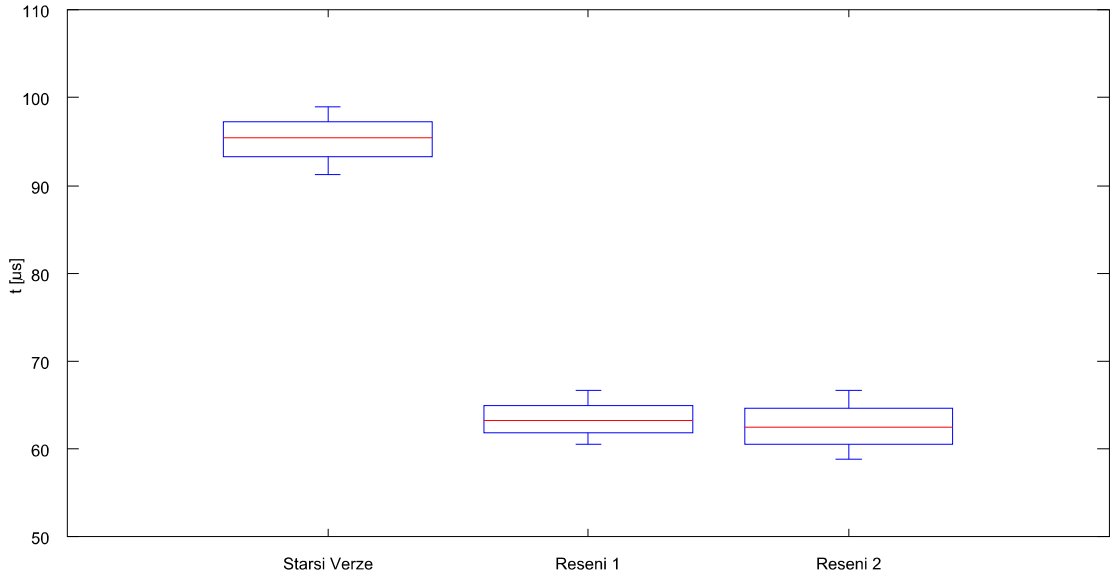
a způsobu jakým s nimi aplikace pracuje.

V prvním grafu (viz. Graf 7.7) jsou výsledky měření doby potřebné k načtení seznamu přístrojů. Je zde vidět více než 4 násobné zlepšení v rychlosti načítání v podání nových řešení oproti tomu starému. Zisk je způsoben hlavně použitím nových synchronizačních metod, konkrétně metody R-W lock, která zohledňuje, zda se přistupuje k datům za účelem čtení nebo se záměrem zapisovat nové hodnoty. Obě nové metody jsou v tomto testu srovnatelné, jelikož obě načítají seznam přístrojů stejným způsobem.



Graf 7.7: Doba potřebná k načtení seznamu přístrojů

Na následujícím grafu (viz. Graf 7.8) jsou hodnoty naměřené při načítání seznamu cyklických operací. V porovnání s předchozím testem je patrné, že nárůst rychlosti u této metody není tak markantní. Tento jev je způsoben nižší komplikovaností vnitřního řešení kódu, který načítání zajišťuje, není tak možné docílit vyššího zisku. I přesto jsou nová řešení struktury aplikace přibližně 1,5× rychlejší než staré řešení. V grafu je dále patrné, že metody dosahují téměř shodných výsledků. Drobný rozdíl je způsoben drobnými odlišnostmi v jednotlivých strukturách, které se nachází i v oblasti seznamu operací.



Graf 7.8: Doba potřebná k načtení seznamu operací

7.4 Shrnutí

Ze všech provedených testů není úplně jednoznačné, která z nových struktur vykazuje vyšší výkonnost. V některých testech vykazuje mírně lepší parametry druhé řešení a v jiných zase to první. Já jsem vzhledem k nižší paměťové náročnosti prvního řešení a také, z mého hlediska, k relativně přehlednější a jednodušší struktuře kódu a do jisté míry také k jeho univerzálnosti v případech možného budoucího rozšíření rozhodl ve finální podobě použít Řešení 1. Toto řešení využívá centrálního správce operací s jedním časovačem.

7.5 Možnosti budoucího rozvoje

V současnosti je systémová služba spolu s její klientskou aplikaci provázána s vizualizační aplikaci ENVIS společnosti KMB systems, s.r.o. formou společného instalátoru. Prostřednictvím aplikace ENVIS je dále možné nastavovat přes systémové registry některé parametry mé aplikace, hlavním z nich je nastavení jazyka, ve kterém aplikace komunikuje s uživatelem. V blízké budoucnosti bude provedena integrace ještě na vyšší úrovni, kdy by měla být grafická část aplikace zvaná client integrována do ENVIS v určité formě pluginu. Následně by pak bylo umožněno spouštět tuto část aplikace přímo z uvedeného ENVISu.

Způsob použití knihoven od KMB, které slouží ke komunikaci s přístroji je

natolik univerzální, že je možné bez větších problémů doplňovat nové typy přístrojů. S novými přístroji může být také spojena potřeba nových typů operací. Díky návrhu nového systému správy těchto operací, nabízí služba větší univerzálnost s ohledem k různým novým typům a jejich případná implementace by neměla tvořit žádný velký problém. Další oblastí, ve které je možné dále pracovat je rozšíření podpory SNMP, jehož stávající stav je popsán v následující kapitole. Například doplnění dalších dat a informací, které jsou prostřednictvím SNMP poskytovány. Stav aplikace rozhodně není konečný a na jejím vývoji se určitě bude i nadále pracovat.

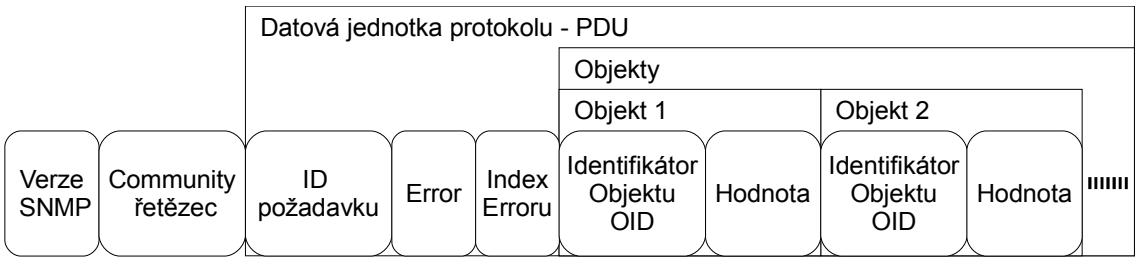
8 SNMP

Jedním z dalších úkolů, vyjma analýzy a návrhu nové struktury systémové služby, bylo doplnit aplikaci o technologii umožňující vzdálený monitoring a správu. Vzhledem k tomu, že těchto rozhraní není velké množství, výběr byl v celku snadný. Rozhodl jsem se použít pravděpodobně nejznámější a nejrozšířenější technologii s názvem Simple Network Management Protocol (SNMP). Jedná se o protokol, který využívá většina síťových zařízení, od tiskáren po routery. Umožňuje monitorování stavu sítě a zařízení v ní umístěných. Roli serveru, který zadává příkazy nebo žádá o data plní SNMP Manager. Se zařízeními, plnícími roli klienta nazývající se SNMP Agents, komunikuje prostřednictvím zpráv. Protokol existuje již ve 3 verzích s názvy SNMPv1, SNMPv2 a SNMPv3. Jednotlivé verze se odlišují hlavně v podpoře zabezpečení a šifrování.

8.1 Popis protokolu

8.1.1 Formát zprávy a její předávání

Zpráva je sestavena z několika částí (viz. Obr. 8.1). Těmi hlavními je jakási hlavička a datová jednotka protokolu zvaná PDU. Hlavička obsahuje informaci o verzi SNMP a community řetězec, který je možné použít jako jaké si heslo pro vzájemnou identifikaci agenta a managera. Pro veřejně přístupná zařízení se běžně používá řetězec „public“. PDU je složitá datová struktura složena opět z několika částí. Opět obsahuje podobu hlavičky a seznam s objekty. Takzvaná hlavička je sestavena z identifikátoru požadavku, který je typu integer a slouží hlavně agentovi, aby mohl spárovat svůj dotaz s obdrženou odpovědí. Dále je zde pole error také typu integer, které dle své hodnoty signalizuje, zda došlo k nějaké chybě a případně k jaké. Poslední pole souvisí s předchozím, jedná se o index erroru, který ukazuje na objekt ve zprávě, ve kterém došlo k chybě. Druhou částí PDU je seznam objektů, který je sestaven, jak již název napovídá, z objektů. Objekty obsahují identifikátor zvaný OID a dále také pole s hodnotou.



Obr. 8.1: Formát zprávy SNMP

Každý ze jmenovaných bloků začíná bajtem, který značí typ daného bloku. Od druhé pozice je zakódována délka daného bloku. Toto pravidlo se týká všech polí ve zprávě, počínaje samotnou zprávou přes PDU seznam objektů až po samotné objekty. Například community řetězec s hodnotou public by vypadal v bajtech následovně (viz. Výpis 8.1). Kde 04 označuje objekt typu string, 06 délku zbylých dat, tedy 6 bajtů a zbytek je slovo „public“ v ASCII.

```
04 06 70 75 62 6C 69 63
```

Výpis 8.1: Ukázka community řetězce v bajtech

Datové typy používané ve zprávách jsou definovány standardem ASN.1 (Abstract Syntax Notation One). ASN.1 je nezávislý na použitém programovacím jazyce a tak jsou typy ve všech jazycích stejné. Typy jsou rozděleny na primitivní a složité. Některé vybrané, které jsem při své práci využil jsou v následující tabulce(viz.Tabulka 1) více viz. [5].

Typ	**ID**
Integer	0x02
String	0x04
Null	0x05
Identifikátor objektu (OID)	0x06
Sekcence	0x30
GetRequest	0xA0
GetResponse	0xA2
Trap	0xA4

Tabulka 1: Datové typy SNMP

Například celá zpráva, objekt nebo seznam objektů jsou typu sekvence tedy 0x30. U objektu bez hodnoty pouze s OID se používá typ null (0x05).

Objekty jsou rozlišovány podle již několikrát zmíněného OID. Představuje něco jako cestu ve stromu k příslušnému objektu. Celý strom se nazývá MIB neboli Management Information Base. Ukázka podoby OID je v následujícím výpise (viz. Výpis 8.2). Veškeré OID pro SNMP začínají 1.3. Jednička odpovídá ve stromu uzlu ISO a trojka uzlu Organization. Dále 6 – dod, 1 – internet, 2 – management, 1 – MIB-2, 1 - system, 2 – System ObjectID. Z jednotlivých uzlů se dá například odvodit, že daný objekt se nachází ve správcovské oblasti, že se jedná o systémovou věc a poslední číslo prozrazuje, že jde o objekt, který souvisí se systémovým OID. V případě, kdy by byl tento objekt zaslán ve zprávě s dotazem, odpovědí by bylo OID dotazovaného systému. Každý výrobce nebo systém používající SNMP by tak měl mít v ideálním případě vlastní sekci v MIB stromu. Toho je možné docílit zaregistrováním OID uzlu u příslušného úřadu.

```
1.3.6.1.2.1.1.2
```

Výpis 8.2: Ukázka OID

V mém případě jsem pro svou aplikaci použil OID 1.3.6.1.3.55. Předposlední číslo (3) odpovídá uzlu experimental, do kterého jsem svou aplikaci zařadil. 55 je uzel, který přísluší přímo mé aplikaci.

Zprávy jsou zasílány v paketech přes UDP/IP. Standardním portem pro SNMP je 161 a 162 pro zprávy typu Trap. Trap zpráva je speciální typ zprávy, kterou SNMP agent zasílá na nastavenou adresu i bez dotazu. Používá se například pro upozornění SNMP Managera na změnu stavu sledované události.

8.2 Implementace SNMP

Pro obsluhu SNMP jsem v systémové službě vytvořil zvláštní knihovnu na základě podoby zpráv a způsoby komunikace. Tato knihovna zajišťuje veškerou činnost, která souvisí se SNMP. Služba naslouchá na příslušném portu a čeká na příchozí SNMP zprávy. Po přijetí jsou ze zprávy načteny prostřednictvím různých funkcí potřebné parametry. Na základě načtených parametrů a OID objektů, které zpráva obsahuje je vytvořena příslušná odpověď, která je vrácena tazateli. Služba v současnosti podporuje několik typů servisních zpráv. Mezi ně patří například dotaz na informace o aplikaci,

název aplikace, její umístění a další. Tyto zprávy jsou sjednoceny v OID s číslem 1.3.6.1.2.1.1 a jedná se o skupinu, která sdružuje informace o systému. Dále jsou podporovány zprávy dotazující se na stav SNMP, jako počet chyb, počet obdržených zpráv a podobně. Těmto zprávám přísluší OID 1.3.6.1.2.1.11. Nakonec je služba schopná odpovídat i na dotazy z vlastní sekce MIB s OID 1.3.6.1.3.55. V této části jsou dotazy na seznam přístrojů připojených k systémové službě nebo na vyžádání jednoduché zprávy o probíhajících operacích. Nakonec tato knihovna umí zasílat také Trap zprávy. Tyto zprávy mají vlastní OID a to 1.3.6.1.3.55.5. V systémové službě jsou využívány k upozornění uživatele na změnu stavů některých nastavených měřených parametrů. Dále mohou také informovat o případném restartu nebo novém startu systémové služby.

9 Závěr

Při práci na tomto diplomovém projektu jsem se podrobně seznámil s různými možnostmi analýzy výkonnosti aplikací vytvořených s využitím knihoven .NET v OS Windows. Setkal jsem se, jak s hotovými nástroji umožňující monitorování jednotlivých částí aplikace, tak i s tvorbou vlastních komponent, jejichž účelem bylo sledování další vnitřních parametrů potřebných pro následnou analýzu. Dále jsem rozšířil své znalosti v oblasti tvorby vícevláknových aplikací s použitím různých metod v různých verzích .NET. Celá systémová služba je vytvořena v programovacím jazyce C# s použitím prostředí VS 2010. Vzhledem k tomu, že se C# jsem se setkal již ve své bakalářské práci, kde jsem o něm získal dostatečné znalosti, nemusel jsem se nyní zabývat těmito základy a mohl se věnovat přímo zadaným úkolům. Nové tedy pro mne byly v oblasti programování hlavně metody v rámci nejnovějších verzí .NET.

Výsledná optimalizovaná verze systémové služby si zachovala veškeré nabízené funkce jako její neoptimalizovaná předloha. Původní verze aplikace vykazovala hlavně velkou nehospodárnost s operační pamětí při vykonávání zadaných cyklických operací. Tento problém se mi podařilo vyřešit přepracováním celé struktury a způsobu nakládání s tímto typem operací. To má za následek výraznou úsporu operační paměti, která je nyní využívána pouze v případech, kdy dochází k vykonávání nějaké činnosti některou z operací. Úspora v porovnání s původním řešením je tím větší, čím více zadaných úkolů má systémová služba vykonávat. Dalším významným přínosem optimalizace je snížení vytížení procesoru. I tento přínos byl způsoben hlavně úpravou samotné struktury aplikace, kdy se změnil způsob zacházení s některými parametry a vykonávání některých činností. Dosáhl jsem také navýšení rychlosti při práci s daty sdílenými mezi více vlákny. Použitím nových a komplexnějších metod pro synchronizaci došlo tedy ke zkrácení doby trávené v uzamčené oblasti a tak i k nárůstu škálovatelnosti aplikace.

Nakonec byla systémová služba doplněna o základní podporu technologie SNMP, díky které je možné sledovat stav a činnosti služby i prostřednictvím softwaru, který tento protokol podporuje. Implementace spočívala ve vytvoření vlastní knihovny, která spravuje veškerou činnost v rámci této technologie.

V průběhu prací na tomto projektu jsem narazil na různé zajímavosti a problémy, které jsem popsal v tomto textu. Z hlediska budoucnosti bude systémová služba i nadále

rozvíjena. Rozvoj se bude týkat jak doplňování podpory nových přístrojů, ať už přístrojů společnosti KMB nebo jiných. Je možné také doplnit službu o některé nové druhy operací, v závislosti na nových přístrojích. Dalším směrem rozvoje může být doplnění podpory o různé technologie v oblasti monitorování stavu, jakými byla například implementace SNMP v této práci. V nejbližší době s největší pravděpodobností dojde k plné integraci uživatelské části aplikace do vizualizační aplikace ENVIS společnosti KMB systems, s.r.o. ve formě jakéhosi pluginu. Rozhodně nevylučuji ve vzdálenější budoucnosti také provedení obdobné restrukturalizace aplikace, jaké byla podrobena v této práci. Ne však v takovém rozsahu a pouze v případě, že budou opět dostupné nové a lepší metody v rámci platformy .NET.

Seznam použité literatury

[1] *.NET Framework Developer's Guide* [online].

URL: ⟨http://msdn.microsoft.com/⟩.

[2] VIRIUS, Miroslav. *C# - Hotová řešení*, Computer Press, ISBN 8025110842

[3] BLÍŽKOVSKÝ, Martin. *Vzdálený sběr a archivace dat z měřících přístrojů SMP, Bakalářská práce*

[4] BLÍŽKOVSKÝ, Martin. *Systémová služba pro obsluhu přístrojů SM a NOVAR, Semestrální projekt*

[5] *Simple Network Management Protocol (SNMP)* [online]

URL: ⟨http://www.cisco.com/en/US/tech/tk648/tk362/tech_tech_notes_list⟩.

[6] *C# – Best Practices For Writing High Performance Code* [online]

URL:⟨http://www.csharphelp.com/2010/02/c-best-practices-to-write-high-performance-code/⟩.

[7] FREEMAN, Adam. *Pro .NET 4 Parallel programing in C#*, APress, ISBN 978 - 1430229674

[8] *Category Concurrency Patterns* [online]

URL: ⟨http://c2.com/cgi-bin/wiki?CategoryConcurrencyPatterns⟩.

[9] CHITTURI, Sriram, *A component for event scheduling inside an application* [online]

URL: ⟨http://www.codeproject.com/KB/system/eventscheduler.aspx⟩.

[10] *Dokumentace k měřícím přístrojům a aplikaci ENVIS* [online]

URL:⟨http://www.kmb.cz/⟩

[11] *Dokumentace ke knihovnám DXperience* [online]

URL:⟨http://www.devexpress.com/⟩

[12] *Improving .NET Application Performance and Scalability* [online]

URL:⟨http://download.microsoft.com/download/a/7/e/a7ea6fd9-2f56-439e-a8de-024c968f26d1/ScaleNet.pdf⟩, ISBN 0-7356-1851-8

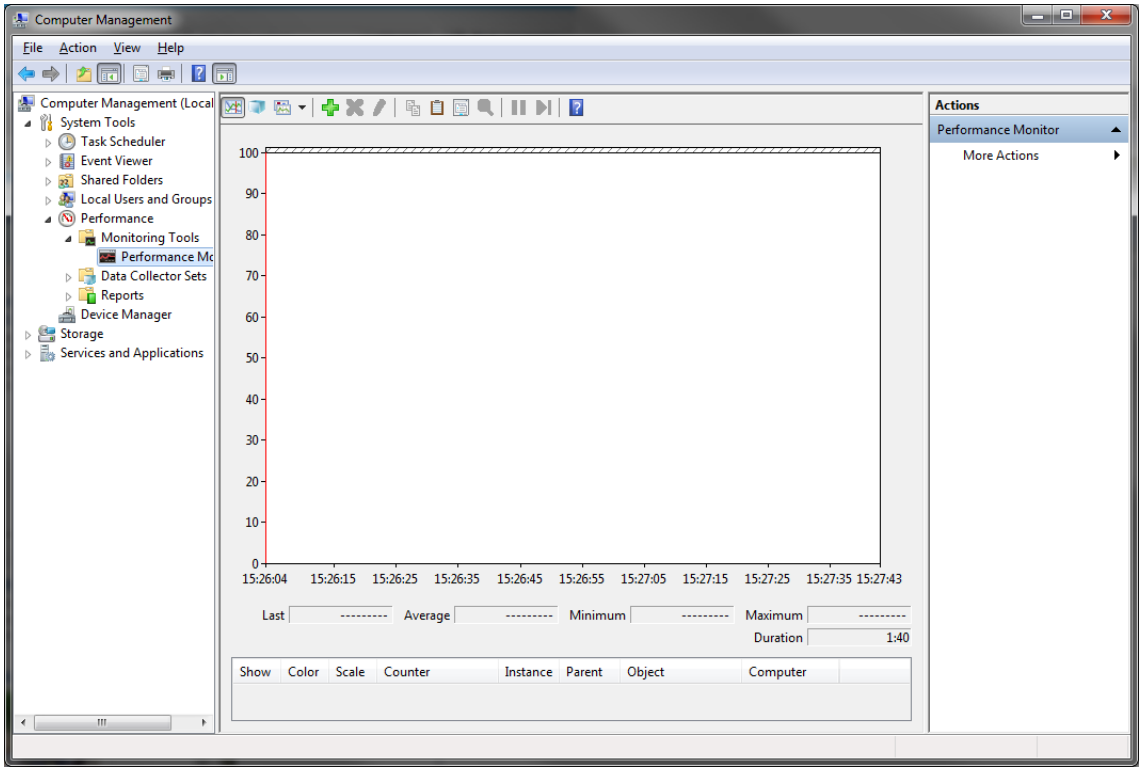
[13] *Microsoft application architecture guide v2.0* [online]

URL:⟨http://download.microsoft.com/download/D/5/9/D599BE85-07EE-4B4D-85D2-FC1C3EDD63B9/Application%20Architecture%20Guide%20v2.pdf⟩,

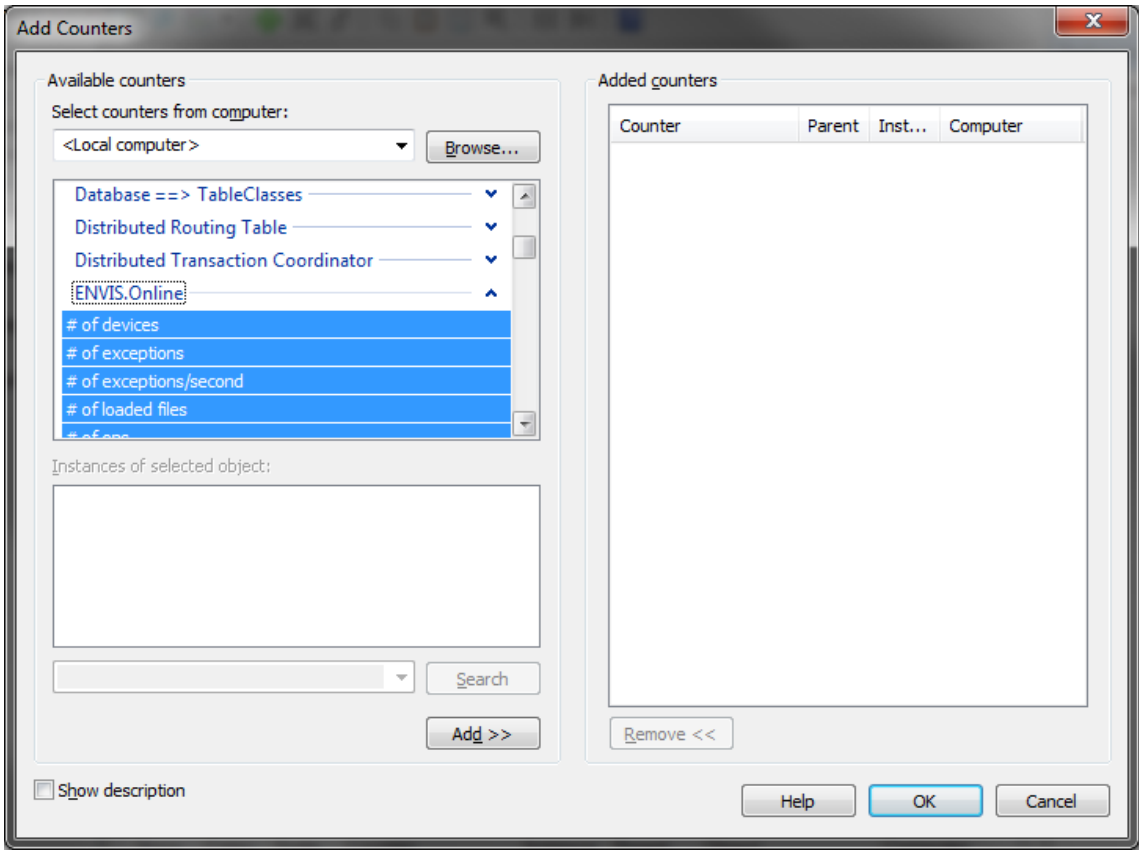
ISBN 978-0735627109

Příloha A – Sledování výkonnostních čítačů

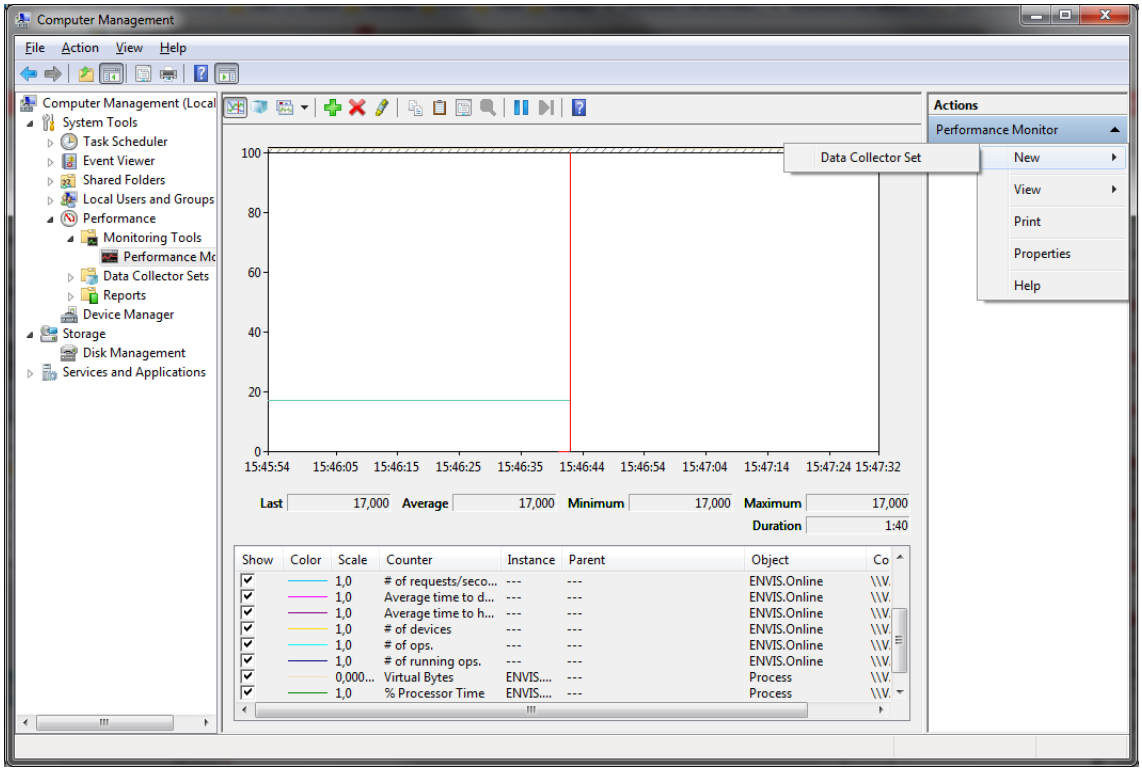
Na následujících obrázcích je znázorněn jeden z možných způsobů sledování a ukládání hodnot výkonnostních čítačů. Na obrázcích je znázorněna metoda zobrazení přes systémového správce.



Obr. A.1: Performance monitor, který zobrazuje hodnoty zadaných výkonnostních čítačů



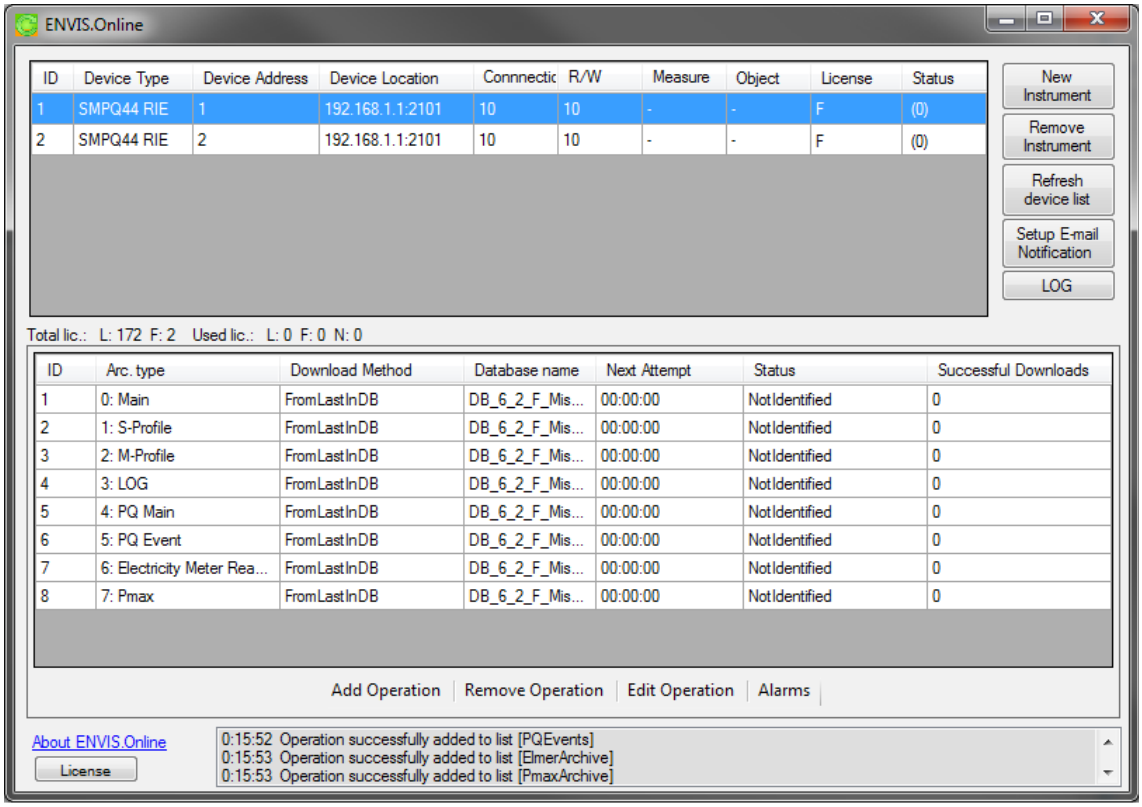
Obr. A.2: Zadání vybraných výkonnostních čítačů, v tomto případě se jedná o vlastní čítače vytvořené za účelem rozšířeného sledování mé aplikace



Obr. A.3: Zobrazení hodnot jednotlivých výkonnostních čítačů prostřednictvím grafu, v pravé části zobrazeno menu umožňující nastavení záznamu naměřených dat do souboru

Příloha B – Klientská aplikace

Systémová služba pracuje na jiné úrovni než běžné aplikace a proto nemá žádné grafické prostředí. V mém případě je absence GUI řešena grafickou aplikací, která slouží jako prostředník v komunikaci mezi uživatelem a službou. Jejich komunikace probíhá formou zpráv prostřednictvím meziprocesové komunikace MSMQ. Na následujícím obrázku je zobrazena podoba této aplikace a dále také podoba zpráv, jejichž prostřednictvím mezi s sebou aplikace komunikují.



Obr. B.1: Ukázka podoby klientské aplikace, jejím účelem je umožnit uživateli nastavování a monitorování běžící systémové služby

```
<Message>
  <MessageHead>
    <ProtocolVersion>1.0</ProtocolVersion>
    <ClientAddress>127.0.0.1</ClientAddress>
    <ClientQName>clientQ_1</ClientQName>
    <ServiceAddress>127.0.0.1</ServiceAddress>
    <MsgID>12</MsgID>
    <Request>ADDDEVICE</Request>
    <Device>1</Device>
  </MessageHead>
  <MessageBody>
    <AvailableDevices>
      <DevIdentNumber>0</DevIdentNumber>
      <TCPSetting>
        <IP>192.168.1.1:12101</IP>
        <DeviceAdress>1</DeviceAdress>
      </TCPSetting>
    </AvailableDevices>
  </MessageBody>
</Message>
```

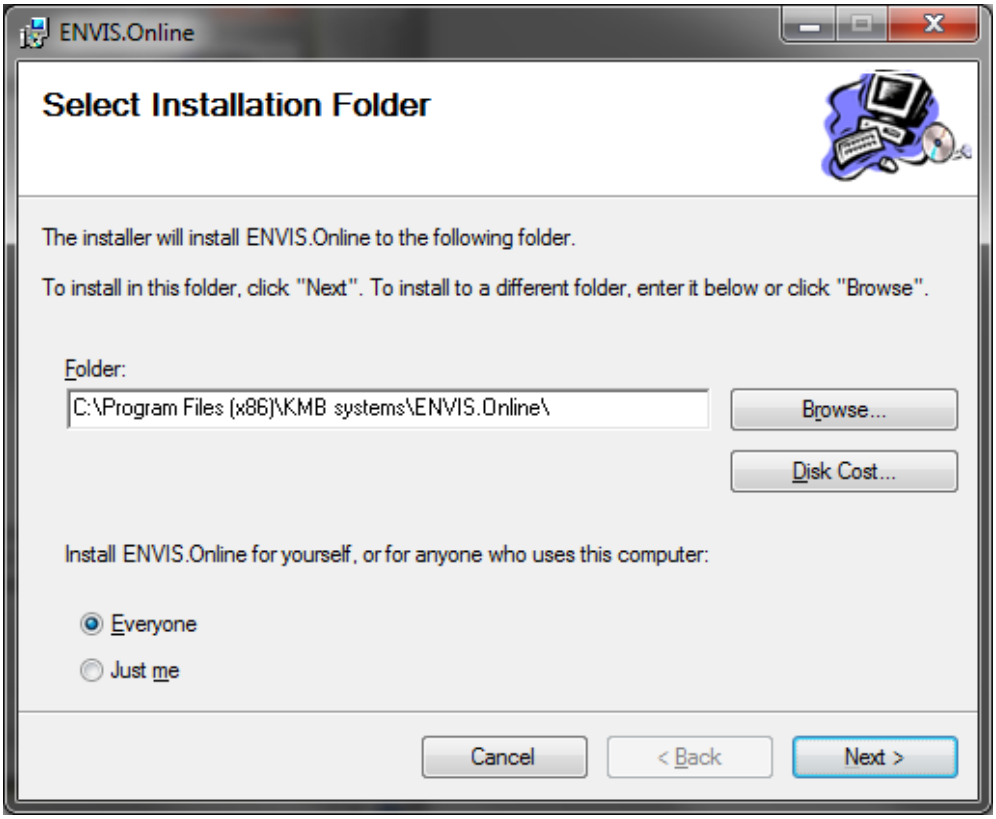
Výpis B.1: Ukázka XML zprávy – systémová služba s klientskou aplikací komunikuje prostřednictvím XML zpráv, každá zpráva se skládá z hlavičky a těla. Zpráva v tomto výpise přikazuje službě přidat požadovaný přístroj do svého seznamu.

Příloha C – Instalace aplikace

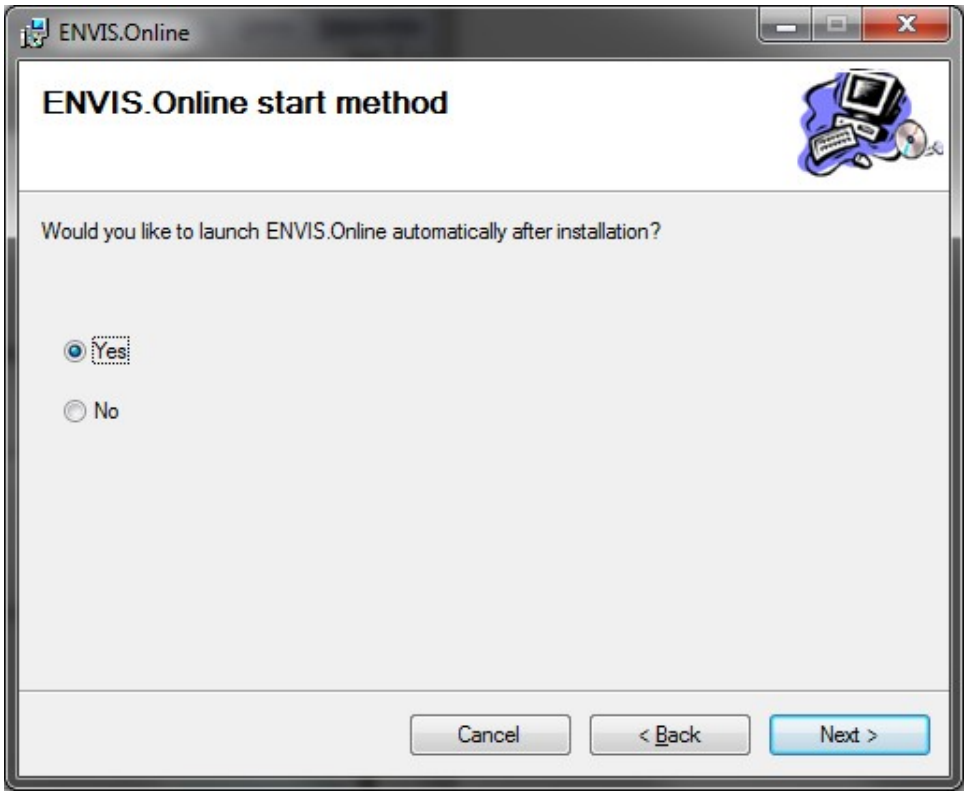
Instalační balíček je vytvořen s použitím šablony *Setup Project,* která se nachází v modulu *Setup and Deployment* v programovacím prostředí Microsoft Visual Studio. Díky tomu jsou tak do instalačního souboru zahrnuty veškeré potřebné knihovny. Vyjma vlastního instalačního balíku může být systémová služba s názvem ENVIS.Online nainstalována i prostřednictvím instalátoru aplikace ENVIS.

Pro správnou činnost musí být operační systém vybaven alespoň SQL Serverem 2005 s rozšiřujícími balíčky a technologií MSMQ. Starší verze aplikace dále potřebuje ke svému chodu rozhraní .NET 2.0, v případě nové optimalizované verze je zapotřebí .NET 4.

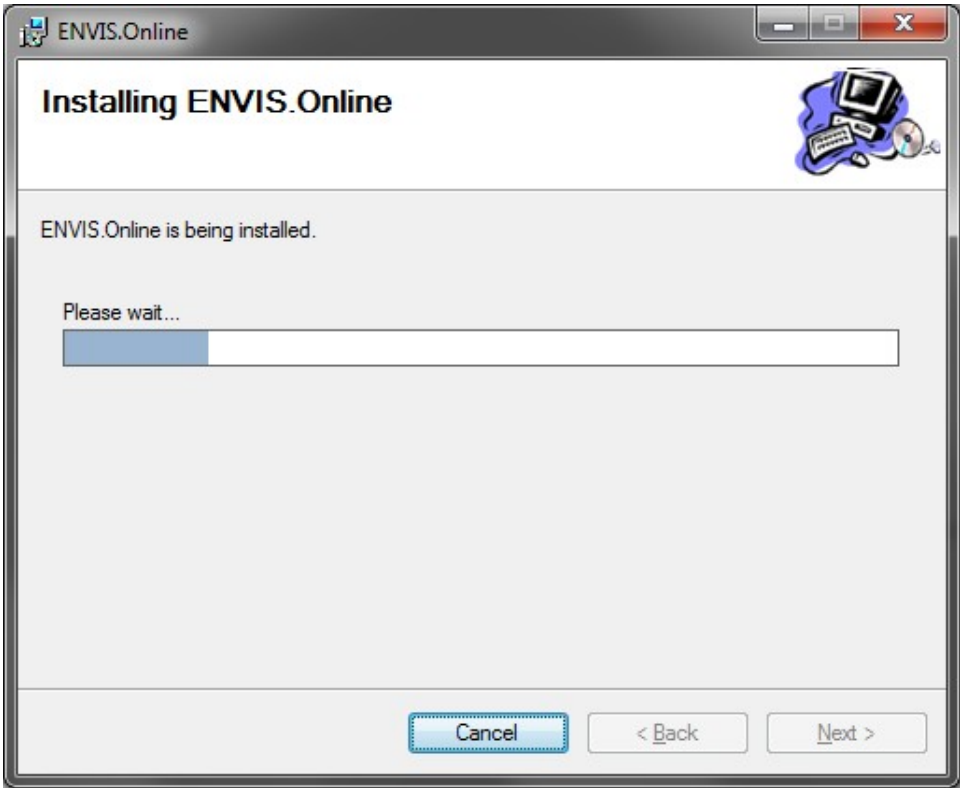
Při použití obou dostupných způsobů instalace je technologie MSMQ a případně také rozhraní .NET nainstalováno automaticky. Navíc u instalátoru aplikace ENVIS jsou při instalaci nainstalovány i potřebné komponenty SQL Serveru.



Obr. C.1: Popis instalace – Výběr umístění



Obr. C.2: Popis instalace – Volba způsobu spouštění systémové služby. Existují dvě možnosti, automatické po startu Windows a ruční.



Obr. C.3: Popis instalace – Zobrazení průběhu instalace