Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Systém pro automatické spouštění Java programů v distribuovaném prostředí

Plzeň, 2014 Yaseen Ali

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně s výhradně s použitím citovaných pramenů.

V Plzni 25.06.2014

Yaseen ALI

Poděkování

Chtěl bych pěkně poděkovat svému vedoucímu práce, panu Ing. Tomáši Potužákovi, Ph.D., za jeho hodnotné rady a odborné vedení během mé práce, a taky za jeho podporu, inspiraci, a za veškerý čas, který mi věnoval. Dále bych chtěl poděkovat svému strýci, panu Hassanu Alimu, ze jeho podporu během mého studia.

Abstract

This Master's work is about creating a system using Java programming language for automatic and repeatetive execution of Java applications in the distributed environment and collect the exceution results on one place.

The system will have server-client architecture, and each side will have a configuration file as input data. The output data will be the execution result, which will be sent as output files back to the server by the clients.

The system will allow the user to execute several tasks at the same time on multiple computers in the distributed environment using one central computer, where the server side of the system will be running and where all execution results will be collected.

Abstrakt

Cílem této diplomové práce je vytvořit systém pro automatické spouštění Java programů a shromáždění výsledů v distribuovaném prostředí. Systém bude napsaný v programovacím jazyce Java.

Systém se bude skládat z dvou částí, serverová část a klientská část. Každá část bude mít svůj konfigurační soubor na vstupu a výstup systému budou soubory výstupů spouštěních aplikací, které klienti budou posílat zpátky serveru.

Systém bude uživateli ušetřit čas tak, že uživatel bude moci spouštět několik Java programů na několika počítačích paralelně v distribuovaném prostředí a výsledky spouštění budou shromážděné na jednom místě.

Obsah

1 Úvod

I když rychlost počítačů neustále roste, stále existuje velké množství úloh vyžadující výpočetní výkon daleko přesahující běžní stolní počítače. Ne každý si může dovolit superpočítač, ale existuje i jiná cesta jak výpočet urychlit. Pokud úlohu vhodně rozdělíme mezi větší množství pomalejších počítačů, můžeme dosáhnout velmi velké výpočetní kapacity.

Cílem této práce je, vytvořit systém pro automatické opakované spouštění existujících Java aplikací a shromažďování výsledků v distribuovaném prostředí. Systém tedy musí být schopen spouštět distribuované aplikace, které jsou přeložené ve formě `.jar` souborů či `.class` souborů na několika počítačích zároveň. Po dokončení výpočtu distribuované aplikace pak systém shromáždí výsledky, které aplikace vložila do souborů a případně vypsala ho na standardním výstupu.

Ideálním příkladem použití systému bude například při simulaci silniční dopravy, kde uživatel bude moci spouštět simulační aplikaci na několika počítačích v distribuovaném systému, pokaždé s jinými parametry a opakovaně kolikrát bude potřeba.

2 Distribuované a paralelní prostředí

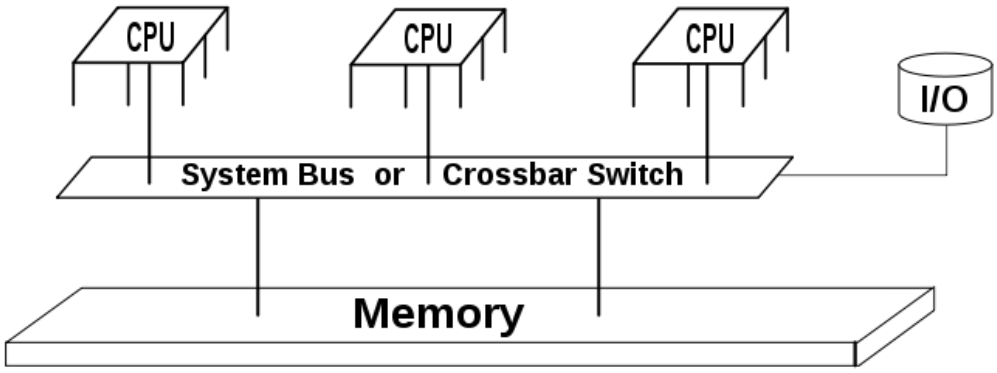
Jak vyplývá z předchozí kapitoly, výpočet lze urychlit využitím více počítačů najednou či více procesorů v jednom počítači najednou. Podle toho, jakou možnost zvolíme, máme k dispozici distribuované či paralelní výpočetní prostředí.

Je důležité zmínit, že je třeba interakci mezí procesy synchronizovat vhodným synchronizačním primitivem. Volba synchronizačního primitiva záleží na technologii, kterou používáme na implementaci aplikace.

2.1 Paralelní prostředí

S víceprocesorovými počítači je možné výpočet urychlit vytvořením vícevláknové aplikace, kde několik vláken vykonává výpočet paralelně, s předem definovanými pravidly, neboli kritérií paralelního programu. Každý program by měl být Flow Correct, což znamená že se program bude chovat deterministicky a končit v konečném čase a dávat stejný výsledek při každém běhu. Každý program by měl být Logically Correct, což znamená, že dává správný výsledek.

Tento způsob urychlení výpočtu se nazývá paralelní výpočet se sdílenou pamětí, kde procesy či vlákna sdílejí stejnou paměť a komunikace mezi nimi probíhá formou sdílení dat. Pro správnou funkci je třeba přístup ke sdílením datům synchronizovat. K tomu se využívají synchronizační primitiva jako jsou zámky, semafory, monitory atd. [1].



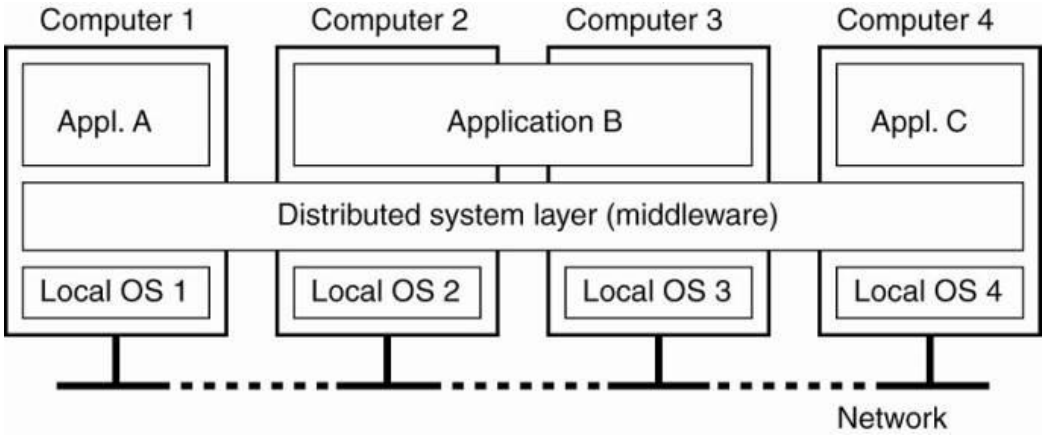
Obrázek 1 - Víceprocesorový systém se sdílenou pamětí (převzato z [2])

Distribuované prostředí

Jako další způsob, jak urychlit výpočet je implementovat paralelní aplikace s distribuovanou pamětí, kde výpočet je paralelně distribuován na několik uzlů po síti. Jeden uzel má 1-n procesorů. Aplikace se tak skládá z několika spolupracujících procesů. Tato forma urychlení výpočtu se nazývá distribuovaná aplikace, protože typicky k dispozici není žádná sdílená paměť. Komunikace mezi spolupracujícími procesy probíhá formou zasílání zpráv [1].

Hlavním cílem distribuovaného výpočetního systému je připojení počítačů transparentním, otevřeným a stupňovatelným (tj. rozšiřitelnost při vzrůstajícím zatížení) způsobem. Takové uspořádání je o hodně odolnější výpadkům a výkonnější než propojené samostatné výpočetní celky [3].

Špatný návrh distribuovaného systému může snížit celkovou spolehlivost systému, kdy porucha jednoho uzlu může působit rušení výpočtu na jiných. Mnoho typů úloh není dobře přizpůsobitelné pro výpočet v distribuovaném prostředí, typicky z důvodu objemu komunikace či synchronizace mezi uzly [3].



Obrázek 2 -Systém s distribuovanou pamětí (převzato z [4])

3 Opakované spouštění Java programů v distribuovaném systému

Při zkoumání existující možnosti pro opakované spouštění Java aplikací v distribuovaném systému, bylo obtížné najít konkrétní příklad, který by odpovídal zadání aspoň částečně.

V následujících podkapitolách jsou popsány některé existující možnosti pro opakované spouštění Java aplikací v distribuovaném prostředí.

3.1 Remote Computing Systems (vzdálené výpočetní systémy)

Obecně existují takzvané vzdálené výpočetní systémy, které poskytují možnost probíhání výpočtů vzdáleně na distribuovaném systému.

Tyto systému se skládají z centrálního CPU a několika vzdálených terminálů, ze kterých uživatel může komunikovat s centrálním CPU a spouštět na něm programy. Což představuje jeden z hlavních konceptů těchto systémů, a to je poskytovat vzdáleným koncům (klientům, aplikacím,...) služby centrálního výkonného serveru [5].

Typickým příkladem jsou dnešní systémy cloudové architektury (obr. 3).

Obrázek 3 - Cloudová architektura (převzato z [6])

3.2 Selenium Framework

V praxi jsou ještě další možné způsoby, jak opakovaně spouštět Java aplikace automaticky a vytvářet různé druhy reportů o výsledcích spouštění. Jako například ve firmě, která vyvíjí a poskytuje softwarové produkty, tak jedna z hlavních fází životního cyklu projektu je fáze testování, které má na starosti oddělení QA (Quality Assurance).

Na testování v Javě se většinou volí Selenium framework, který je velmi mocný framework na psaní testů v jazyce Javy a je hodně populární. Na automatické a opakované spouštění testů psaných v Seleniu se používají různé test managements (řízení testů) systémy, které můžou být interně vyvíjené systémy, a nebo přes koupené licence. Smysl paralelizace v tomto případě spočívá ve spouštění test suite (soubor testovacích případů) na několika VM (Virtuálních strojích), tím se úkoly mezi VM rozdělují a výsledky se shromažďují na jednom místě, tedy tam kde je specifikováno v systému řízení testů [7].

3.3 PVM

Parallel Virtual Machine. Původně pro C a Fortran, dnes existuje pro Javu JPVM. PVM je univerzální výpočetní model pro heterogenní distribuované výpočetní prostředí.

Abstrakce několika různých prostředí, různé operační systémy, různé HW. PVM poskytuje rozhraní, které je umožní využívat jako jeden paralelní počítač. Lze vytvořit i virtuální síť na jednom počítači [8].

U PVM jsou místo vláken procesy, které tvoří distribuovanou aplikaci. Vlákno (Fiber, thread, task) může být vykonávaný programový kód, tj. aktivita v čase, která má svůj stav, který je určen aktuálním místem v programu, obsahem registrů procesoru a obsahem zpracovaných dat. Zatímco proces je dynamická alokace vláken, má stav daný vlákny. Každý proces obsahuje vždy aspoň jedno vlákno, které vytvořil operační systém. Vlastní prostředky, které operační systém přidělil důsledkem činnosti některého vlákna. Vlákna jednoho procesu můžou běžet současně, je-li k dispozici více než jeden procesor [9].

U PVM každý proces má svého démona, který zprostředkovává doručování zpráv, je zodpovědný za spolehlivý přenos a doručení zprávy, zprávu k odeslání vždy přebere démon příslušný k procesu a z tid zjistí kam ji poslat a provádí un/packing zprávy [10].

U PVM spawn je příkaz v konzoli, který spustí primární proces. Pvm_spawn spustí několik procesů podle zadaného programu. Pvm_spawn vrací identifikátory procesů. Lze buď určit kde mají procesy spustit nebo to nechat na rozhodnutí PVM. Jestli má proces PVM běžet na různých platformách, je třeba, aby programátor dal k dispozici verze pro všechny tyto platformy.

3.3.1 JPVM (Java Parallel Virtual Machine)

JPVM je PVM-like knihovna s implementovanými objektovými třídami. PVM je rozhraní pro paralelní programy. Není závislé na implementaci a je tak funkční i mezi počítači s rozdílnými operačními systémy. Jedná se o komunikační vrstvu využívající zejména TCP protokol za účelem spojení výkonů procesorů více počítačů k vytvoření virtuálního superpočítače [11].

JPVM bohužel není interoperabilní se standardním PVM. JPVM je kompletně oddělená od PVM, není interoperabilní implementace. Na druhé straně mince ne interoperability, JPVM podporuje spoustu funkcionalit, které ve standardním PVM nejsou. Například:

- Zprávy v JPVM nejsou nikdy přeposílány přes démony, ale přímo ke cílovému úkolu. Standardní PVM má démony, které běží na pozadí a když se volá pvm_spawn(), démon připraví a alokuje všechny potřebné systémové zdroje na běh úkolu.
- JPVM je thread safe. Na jednom úkolu může spolupracovat několik vláken [11].

3.4 MPI (Message Passing Interface)

Je knihovna určená původně pro C a Fortran, dnes existuje pro Javu mpiJava. MPI je komunikační rozhraní pro paralelní programy. Knihovna s definovaným API:

- Nezávislé implementace, možnost přenositelnosti,
- Možnost optimalizace pro různé HW.

Čistým cílovým prostředím MPI aplikací jsou gridy a víceprocesorové počítače. Dalším prostředím jsou clustry, tj. skupina počítačů propojených sítí, které se navenek tváří jako jeden superpočítač s distribuovanou pamětí. Počítače v clustru spolupracují pomocí jasně definovaného rozhraní [12].

3.4.1 Jádro práce s MPI

Rozhraní MPI poskytuje klíčové funkce:

- MPI_Init: Inicializace MPI, tady proces deklaruje, že bude používat MPI.
- MPI_Comm_Size: Vrací počet procesů používajících MPI
- MPI_Comm_Rank: Vrací vlastní identifikační číslo v MPI. ID se používá pro zasílání zpráv.

- MPI_Send: Odeslání zprávy.
- MPI_Recv: Příjem zprávy.
- MPI_Finalize: Proces deklaruje, že už nebude používat MPI. Úklid a rozlučka s MPI rozhraním [12].

3.4.2 mpiJava

Třída MPI má jenom statické členy. Chová se jako modul, obsahující globální služby, jako například MPI_Init a spousta dalších globálních konstant včetně defaultního komunikátoru COMM_WORLD.

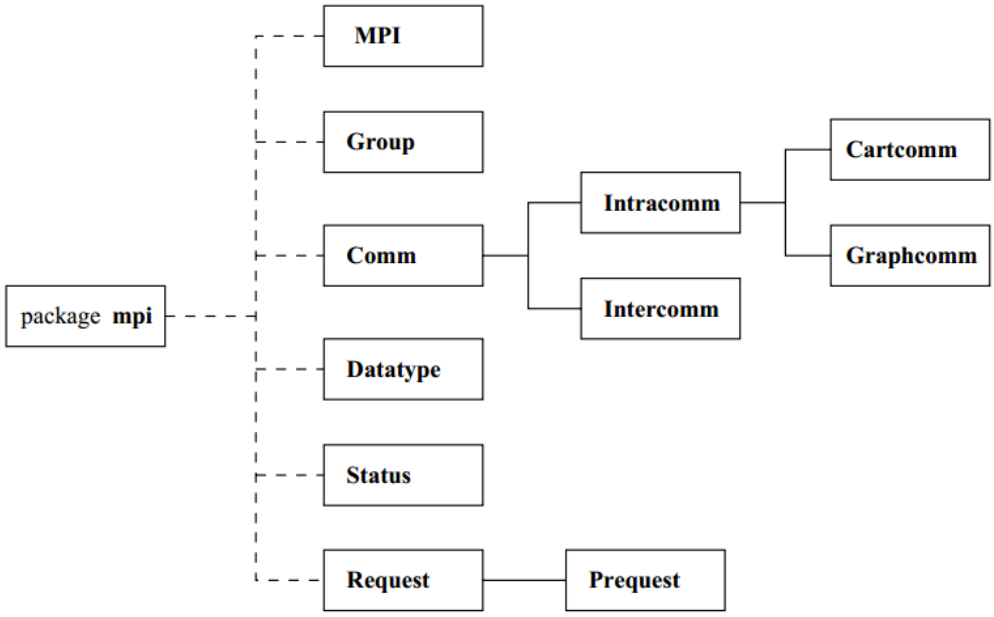
Hlavní třídy mpiJava jsou na obr. 4. Nejdůležitější třída balíku je třída komunikátoru Comm. Všechny komunikační finkce v mpiJava jsou členy Comm třídy nebo jejich sub-tříd [13].

Jako další důležitá třída je třída `Datatype`, tato třída popisuje datový typ prvků zprávy připravenou na odeslání. Různé základní datové typy (viz tab. 1) jsou předdefinovány v balíku a korespondují k primitivním datovým typům Javy:

Tabulka 1Table 1 - mpiJava datové typy (převzato z [13])

MPI datatype	Java datatype
MPI.BYTE	Byte
MPI.CHAR	char
MPI.SHORT	Short
MPI.LONG	Long
MPI.INT	Int
...	...

Hlavní třídy mpiJava:



Obrázek 4 - mpiJava třídy (převzato z [13])

3.4.3 Speciální vlastnosti mpiJava

mpiJava API je modelována co nejvíce podobně jako MPI C++, některé změny ale byly nutné, jako například u list argumentů, tj. v Javě nemůžeme předávat argumenty s pomocí jejich reference. Obecně výstup mpiJava metod je daný návratovou hodnotou funkcí.

Ve většině případů MPI funkce vrací více než jednu hodnotu/jeden výsledek. Tenhle fakt je v Javě ošetřen několika způsoby. Občas MPI funkce inicializuje list elementů a vrací počet modifikovaných elementů z původních inicializovaných elementů z listu. V Javě typicky vracíme list výsledků a vynecháváme počet elementu listu, počet můžeme získat postupně z velikosti listu výsledků [13].

Občas MPI funkce inicializuje objekt a vrací flag (tag), jako signál, jestli operace byla úspěšná či ne. V Javě vracíme objekt jako null v případě, že operace nebyla úspěšná.

Normálně v mpiJava, MPI destruktory jsou zavolány voláním Java `finalize` metod třídy. Java carbage collector v tom hraji hlavní roli. Proto pro spoustu tříd není třeba používat MPI_class_FREE funkci. Výjimky jsou třídy `Comm` a `Request`, které mají explicitní Free členy [13].

Jednoduchý příklad v mpiJava je uvedený na následujícím výpisu 1:

```
import mpi.*;
class Hello {
      static public void main(String[] args){
            MPI.Init(args);
            int myrank = MPI.COMM_WORLD.Rank();
            if(myrank == 0)}
                  char [] message = "Hello, there".toCharArray() ;
                  MPI.COMM_WORLD.Send(message,0,message.length,  MPI.CHAR,
            1, 99);
            } else {
                  char [] message = new char [20] ;
                  MPI.COMM_WORLD.Recv(message, 0, 20, MPI.CHAR, 0, 99) ;
                  System.out.println("received:" + new String(message) +
            ":");
            }
            MPI.Finalize(); }
   }
```

Výpis 1 – příklad mpiJava (převzato z [13])

3.5 Hodnocení

podle testů výkonnosti, co provedly studenti z School of Computer Science, University of Portsmouth, Southsea, Hants, UK, PO4 8JF, mpiJava API musí stačit MPI programátorům, a to nejen ze hlediska funkcionalit, ale taky z hlediska dobré výkonnosti ve srovnání s podobným MPI programem napsaným v C [13].

Oba PVM a MPI mohou běžet v heterogenním prostředí, ale musí to umožňovat jejich implementace a implementace aplikace.

Sice MPI nabízí jednodušší a efektivnější abstrakce na vyšší úrovni, které umožňuje uživateli definovat datové struktury a ty pak přenášet ve zprávách a nabízí také bohatší možnosti pro přenos zpráv, ale pro implementaci systému se nehodí a není vhodné ji využít [14].

To samé pro PVM, které umožňuje programátorovi napojit se do systému pomocí nízko-úrovňových rutin PVM, ale stejně se nehodí pro implementaci systému [14].

Selenium Framework je specifický hlavně pro testování softwarů, a vzdálené výpočetní systémy mají širší uplatnění než cíl práce.

Z toho plyne, že ani jednu ze zmíněných možnosti nebudeme používat a na vytvoření systému bude potřeba mít vlastní implementaci.

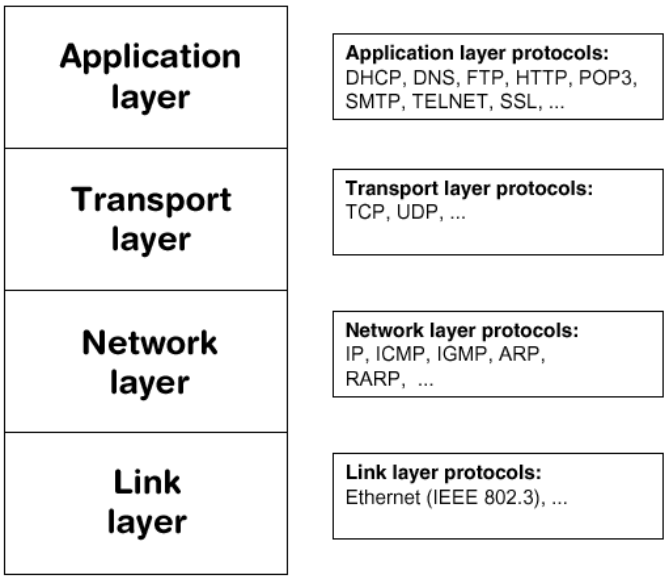
4 Síťová komunikace v Javě

V následujících podkapitolách budou popsány některé technologie na síťovou komunikaci v programovacím jazyce Java.

4.1 TCP/IP

To je Transmission Control Protocol/Internet Protocol, je základním komunikačním jazykem nebo protokolem internetu. TCP/IP je dvouvrstvý program. Vyšší vrstva, Tramission Control Protocol, stará se o rozsekání zprávy nebo souboru do menších paketů, které se budou posílat po síti a přijaty na druhé straně TCP vrstvou, která je vrátí do původního stavu. Nižší vrstva je Internet Protocol, která se stará o adresování jednotlivých posílaných paketů, takže každý paket se dostane tam kam má [15].

TCP/IP používá model klient-server pro síťovou komunikaci, kde jeden uživatel (klient) žádá o službu serveru, a server mu ji poskytne. Detailní struktura TCP/IP protokolu je popsána na obr. 5.



Obrázek 5 - TCP/IP struktura (převzato z [17])

4.1.1 Popis jednotlivých vrstev TCP/IP protokolu

- linková vrstva, neboli vrstva síťového rozhraní: Nejnižší vrstva umožňuje přístup k fyzickému přenosovému médiu. Je specifická pro každou síť v závislosti na její implementaci. Příklady sítí: Ethernet, Token ring, ... [17].

- Síťová vrstva: Vrstva zajišťuje především síťovou adresaci, směrování a předávání datagramů [17].

- Transportní vrstva: tato vrstva je implementována až v koncových zařízeních (počítačích), a umožňuje proto přizpůsobit chování sítě potřebám aplikace. Poskytuje transportní služby kontrolovaným spojením spolehlivým protokolemTCP (transmission control protocol) nebo nekontrolovaným spojením nespolehlivým protokolem UDP (user datagram protocol) [17].

- Aplikační vrstva: největší vrstva síťové architektury internetu, její protokoly definují a specifikují pravidla komunikace a formáty datových struktur pro jednotlivé síťové služby. Aplikační protokoly používají vždy jednu ze dvou základních služeb transportní vrstvy: TCP nebo UDP, případně obě dvě (např. DNS). Pro rozlišení aplikačních protokolů se používají tzv. porty, což jsou domluvená číselná označení aplikací. Každé síťové spojení aplikace je jednoznačně určeno číslem portu a transportním protokolem (a samozřejmě adresou počítače) [17].

4.2 JMS (Java Messaging Services)

JMS je Java API, které umožňuje aplikacím (konkrétně klientům, protože to je peer-to-peer systém) tvořit, posílat, přijímat a číst zprávy. Cílem JMS je poskytnout co nejvíce konceptů a rozhraní v takové podobě, aby se minimalizoval počet konceptů, které musí programátor Javy umět, aby mohl pracovat s MOM (Message Oriented Middleware) a zprávami. Programátor se díky JMS nemusí starat o technické pozadí za koloběhem zpráv [18].

JMS není emailové API, přestože funguje na podobném principu. Základem komunikace mezi klienty v rámci JMS jsou asynchronně posílané zprávy (požadavky, události, odpovědi atd.). Tyto zprávy jsou vytvářeny a užívány aplikacemi a obsahují informace, díky kterým tyto aplikace fungují, nastavují se a koordinují svou činnost v rámci celku [18].

4.3 Java Websocket API

U Websocket aplikace, server zveřejňuje WebSocket koncový bod (end-point) a klient potom používá URI koncového bodu na spojení se serverem. Po zahájení spojení mezi klientem a serverem, Websocket protokol se stane symetrickým protokolem [19].

Server a klient si mohou posílat zprávy mezi sebou po celou dobu spojení, a mohou ukončit spojení kdykoliv budou chtít. Klient se spojí obvykle jenom s jedním serverem, server přijímá spojení od více klientů (multithreading) [19].

Moderní webové prohlížeče implementují Websocket protokol a poskytují Javascript API pro připojení s koncovými body serveru, posílání zpráv a přiřazení zpětné volání metod pro události Websocketu [19].

4.4 Java RMI

Java RMI (Remote Method Invocation) vzdálené volání metod je systém, umožňující objektu, běžící na jednom JVM volat metody jiného objektu, běžící na jiném JVM.

RMI aplikace většinou dají dohromady dva separátní programy, klient a server program. Typický program serveru vytvoří několik vzdálených objektů (remote objects), a nastaví je jako přístupné objekty a čeká na klienty, až se mu ozvou a zavolají metody na těch vytvořených objektech. Typický program klienta získá reference na jeden nebo více vzdálených objektů na serveru a pak zavolá metody na nich [20].

RMI poskytuje obousměrný komunikační mechanismus mezi serverem a klientem, že si mezi sebou posílají data a informace. Takové aplikace se nazývají distributed object applications (aplikace distribuovaných objektů) [20].

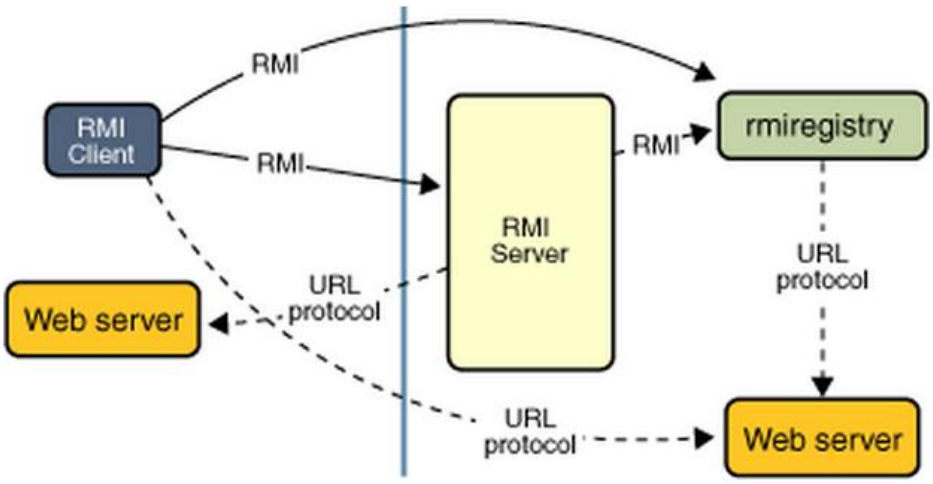
Aplikace distribuovaných objektů potřebuje dělat následující:

- Lokalizovat vzdálené objekty: aplikace můžou používat různé mechanismy na získávání referencí na vzdálené objekty. Například, aplikace může registrovat své vzdálené objekty jednoduchým pojmenovacím mechanismem, který má RMI, registry RMI. Jako alternativní způsob, aplikace může posílat a získávat reference na vzdálené objekty jako součástí vzdáleného volání metod [20].

- Komunikovat s vzdálenými objekty: detaily komunikace řeší RMI, z pohledu programátora jde o klasické volání metod [20].

- Načítat definice tříd objektů, které se posílají mezi aplikacemi: jelikož RMI umožňuje to, aby se objekty posílaly mezi aplikacemi, tak RMI poskytuje taky mechanismus jak načítat definici třídy objektu a jeho data [20].

Následující obr. 6 ilustruje RMI distribuovanou aplikaci, která používá RMI registry pro získání referenci na vzdálené objekty.



Obrázek 6 - Struktura RMI aplikace (převzato z [20])

Server volá registry pro asociaci jména s vzdáleným objektem. Klient si najde vzdálený objekt podle jména v registrech serveru a volá jeho příslušnou metodu.

Na obrázku je taky vidět, že RMI používá existující webový server pro načítání definice tříd objektů ze serveru na klienta a opačně když je to třeba [20].

Jedna z hlavních, centrálních a unikátních vlastností RMI je jeho schopnost stahovat definici třídy objektu, není-li definována na JVM příjemce. Tedy, všechny typy a chování objektu, který byl původně přítomen jenom na jednom JVM, může být dynamicky přenesen na jiné, možno vzdálené, JVM. Chování objektu se nemění, když se přesouvá na jiné JVM, protože RMI pracuje s objekty přímo s jejich aktuálními třídami. V tomto případě jde jenom o rozšíření chování aplikace v distribuovaném systému [20].

Jako každá jiná Java aplikace, RMI distribuovaná aplikace je vytvořena z rozhraní a tříd. Rozhraní deklaruje metody. Třídy implementuje deklarované metody rozhraním, a občas deklaruje další metody taky.

Objekt se stane vzdálením objektem implementací remote interface (vzdáleného rozhraní), které má následující charakteristiky:

- Vzdálené rozhraní rozšiřuje (extends) *java.rmi.Remote* rozhraní,
- Každá metoda z rozhraní deklaruje ke každé aplikační vyjímce (application-specific exception) java.rmi.RemoteException ve své throws sekci.

RMI se chová k vzdálenému objektu jinak než k ne-vzdálenému objektu, když se přesouvají z jednoho JVM do druhého. Místo kopírování implementace vzdáleného objektu do přijímajícího JVM, RMI používá vzdálený stub vzdáleného objektu. Stub se chová jako lokální zástupce, nebo proxy pro vzdálený objekt. Stub se získá jako výsledek volání metody vzdáleného objektu nebo přes RMIRegistry. Klient používá pro odkazy na vzdálený objekt proměnné typu rozhraní, které vzdálený objekt implementuje. Ve skutečnosti, tyto odkazy odkazují na stub příslušného vzdáleného objektu [21].

Klient volá metody na lokálním stubu, který je potom zodpovědný o přeposílání volání metody na příslušný objekt.

Následující je příklad jak vytvářet aplikaci s vzdáleným objektem.

1. Definice rozhraní vzdáleného objektu

Rozhraní vzdáleného objektu musí dědit z rozhraní *java.rmi.Remote*. Metody musí být deklarovány jako vyhazující výjimku RemoteException (vyhazuje ji RMI runtime při problémech komunikace s implementací objektu)

```
interface MyInterface extends Remote {
   int MyOp(int myParam1, char MyParam2) throws RemoteException;
   ...
  }
```

2. Implementace vzdáleného objektu

```
class MyImpl extends UnicastRemoteObject implements MyInterface {
  /* definice konstruktoru a všech metod MyInterface */
 }
```

3. Vygenerování stubu a skeletonů pomocí překladače rmic

   - vstupem rmic je class soubor implementace vzdáleného objektu.
   - výstupem jsou class soubory tříd stubu a skeletonu pro zadaný vzdálený objekt.

4. Implementace server vytvářejícího instance vzdáleného objektu:

   - vytvoření instance serverového objektu a jeho zveřejnění (export). Po exportování začne objekt přijímat požadavky (na anonymním portu). Pokud je bází implementace třída UnicastRemoteObject (nejčastěji) nebo Activatable, provádí se export automaticky, jinak explicitně voláním UnicastRemoteObject.exportObject().
   - nutné nastavit security manager (System.setSecurityManager(new RMISecurityManager());)
   - registrace serverových objektů v registry:

```
MyImpl mi=new MyImpl();
Naming.bind("MyName",mi);
```

5. implementace klienta používajícího vzdálený objekt

- klient deklaruje proměnnou typu rozhraní vzdáleného objektu a vyhledává dříve zaregistrovaný objekt v registry, aby získal jeho stub:

```
MyInterface m=(MyInterface)
Naming.lookup("rmi://host_running_registry/MyName");
```

- po získání stubu již další práce s objektem shodná jako s běžným lokálním objektem:

```
int res=m.myOp(1,'a');
```

6. umístění příslušných tříd na klienta a server
    - Na serveru: stuby, skeletony, implementace, interface
    - Na klientu: stuby, interface

4.5 CORBA

Common Object Request Broker Architecture a RMI jsou nejrozšířenější a nejpoužívanější systémy distribuovaných objektů. CORBA implementace je známa jako ORB (Object Request Broker). Existuje na trhu několik CORBA implementací jako například VisiBroker, ORBIX a další. JavaIDL je další implementace, která přišla jako jádrový balík s JDK1.3 a vyšší. CORBA je implementovaná architektura skupinou řízení objektu (Object Management Group - OMG) [22].

Distribuovaný objekt je definovaný pomocí softwarového souboru, podobný souboru vzdáleného rozhraní u Java RMI. CORBA rozhraní je definované pomocí univerzálního jazyka se zřetelnou syntax, známou jako CORBA Interface Definition Language (IDL) [22].

CORBA byla modelována tak, aby byla platformně i jazykově nezávislá. Proto COBRA objekt může běžet na jakékoliv platformě, kdekoliv po síti, a může být napsaný v jakémkoliv programovacím jazyce, který má mapování na IDL (Interface Definition Language) [22].

Syntaxe COBRA IDL je podobná syntax Javy a C++. Nicméně, objekt definovaný v COBRA IDL souboru, může být implementován ve spoustě dalších programovacích

jazycích včetně C, C++, Java, COBOL, Ada, Lisp, Python. Pro každý z těchto jazyků, OMG má standardní mapování z COBRA IDL do příslušného programovacího jazyka [22].

Jako u Java RMI, COBRA distribuovaný objekt je lokalizován pomocí reference objektu. Jelikož COBRA je jazykově nezávislá, tak její reference objektu je abstraktní entita mapována na reference objektu ve specifikovaném jazyce pomocí ORB [22].

Pro interoperabilitu, OMG specifikuje pro COBRA abstraktní reference objektu protocol Interoperable Object Reference (IOR).

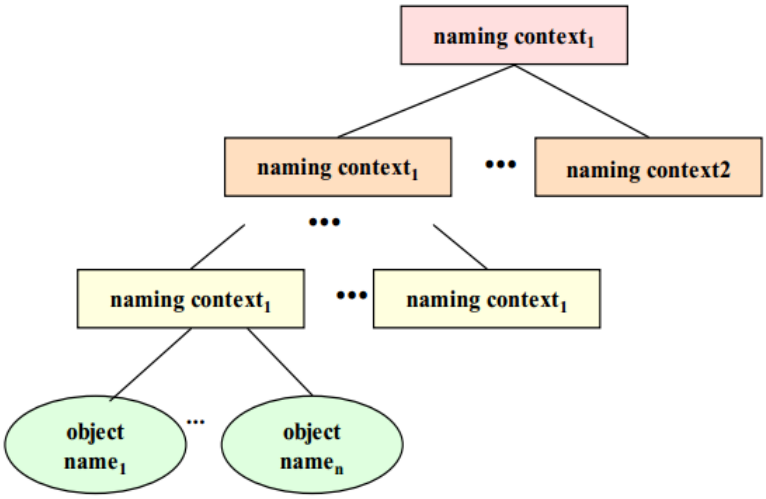
IOR protocol je string obsahující kódování pro následující informace:

- Typ distribuovaného objektu, adresa, kde můžeme distribuovaný objekt najít,
- Číslo portu na server pro distribuovaný objekt a klíč distribuovaného objektu, tj. řetězec bajtů identifikujících objekt [22].

4.5.1 CORBA naming service

COBRA specifikuje generický adresáš služeb. Služba Naming Service slouží jako adresář pro COBRA objekty, a umožňuje ORB klientům získat reference na distribuovaný objekt, se kterým chtějí pracovat [23].

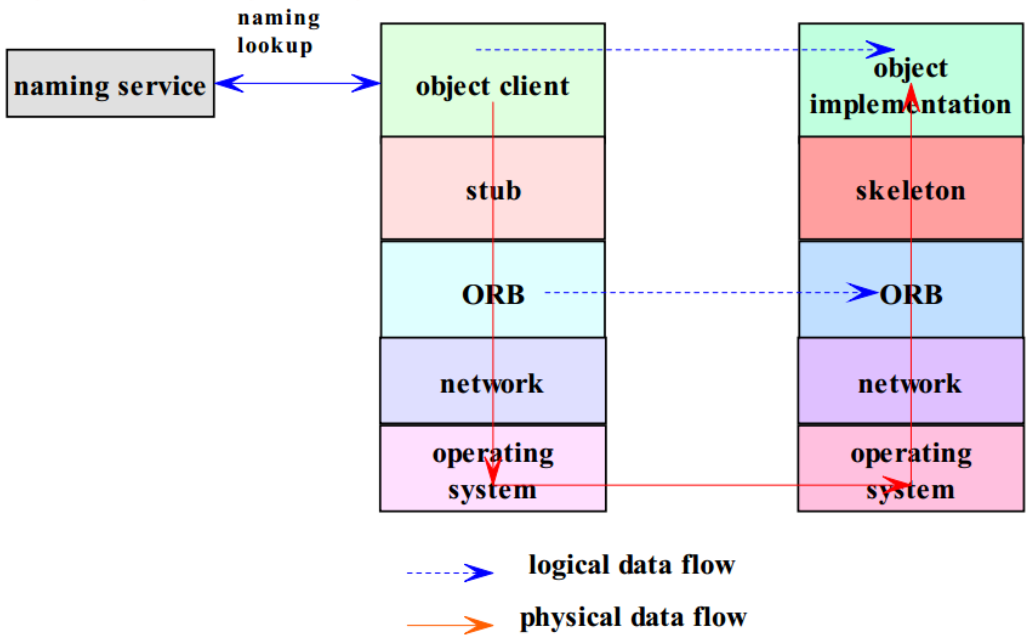
Když klient chce získat reference na nějaký distribuovaný objekt, žádá o to Naming Service, která prohledá příslušný objekt podle jeho jména a vrací klientovi požadovanou referenci. Na následujícím obr. 7 je popsána hierarchie Naming service



Obrázek 7 - COBRA naming service [23]

4.5.2 Základní architektura

Základní architektura CORBA je popsána na obr. 8. CORBA je hodně široké téma, pro zájemce o další detaily, je k dispozici spousta elektronických zdrojů se zajímavými příklady.



Obrázek 8 - COBRA základní architektura (převzato z [23])

4.6 Hodnocení

RMI a CORBA jsou vysokoúrovňové technologie na síťovou komunikaci v Javě, každá z nich ale má výhody a nevýhody.

RMI:

Tabulka 2- výhody a nevýhody RMI (převzato z [24])

výhody	nevýhody
Přenositelné mezi platformami	Jenom platformy, které podporují Javu
Může produkovat nový kód cizímu JVM	Bezpečnostní překážky s vzdáleným spuštěním objektů, omezení ve funkcionalitách kvůli bezpečnostním omezením
RMI je k dispozici od JDK1.02. Tedy, programátoři už mají v tom nějakou praxi.	Naučit se RMI je obtížné pro začátečníky.
Existující system může použít RMI, jelikož přepnutí na jinou technologii může stat hodně času a peněz	Jenom systémy, které podporují Javu

CORBA:

Tabulka 3 - výhody a nevýhody COBRA (převzato z [24])

výhody	nevýhody
Služby mohou být napsány ve spoustě různých jazycích, spuštěné na mnoha různých platformách a přístupné jakémukoliv jazyku s IDL mapováním	Popis služeb vyžaduje použití IDL, které se musí naučit. Implementace nebo používání služeb vyžaduje IDL mapování. Toto všechno stojí čas a peníze na začátku.
S IDL, implementace je pěkně oddělena od rozhraní. Programátor může vytvořit několik různých implementací toho samého rozhraní.	IDL mapovací nástroje vytvoří kód stubů na základě rozhraní. Může se stat, že některé nástroje nebudou integrovat nové změny do existujícího kódu.

COBRA podporuje primitivní datove typy a spoustu datových struktur jako parametry.	COBRA nepodporuje transformace objektu ani kódu.
COBRA je ideálně vhodná pro použití se staršími systémy (legacy systems) a pro zajištění, že aplikace, které momentálně fungují, budou přístupné i v budoucnu.	Budoucnost není určitá – Pokud se COBRA nepodaří dosáhnout dostatečné přijetí průmyslu, její implementace se stanou staršími systémy.
COBRA představuje jednoduchý způsob jak spojit objekty a systemy dohromady.	Pořád je nutné školení, a specifikace COBRA je stale ve stavu toku.
COBRA system můžou poskytovat super výkon	Ne všechny třídy aplikace potřebují real-time výkon, a rychlost system může být jenom obchodní záležitost oproti snadnosti použití čistě javovské aplikace.

Co se tyče dalších již zmíněných technologií, tak JMS je vhodné pro Java EE aplikace, což není cílem naší práce stejně je to s Java WebSocket API.

TCP/IP technologie je v tomto případě nejvhodnější volba na implementaci systému, hlavně kvůli rychlosti TCP/IP vůči ostatním technologiím jako RMI nebo CORBA. TCP/IP vlastní systém adresování, který je zabudovaný do mechanismů fungování, umožňuje identifikovat uzly (zařízení) i dílčí entity (služby atd) bez znalosti detaily jejich připojení. Další příčina úspěchu TCP/IP je jeho dobrá škálovatelnost, původní řešení vzniklo pro sítě s desítkami uzlů, dnes funguje pro internet s milióny uzlů [15].

5 Analýza

V následujících podkapitolách budou popsané specifikace požadavků systému, případy užití a na konci budou popsány použité technologie, pomocí kterých jsem naimplementoval svůj systém.

5.1 Specifikace požadavků

Systém musí být schopen spouštět existující Java aplikace, které jsou přeložené ve formě `.class` či `.jar` souboru a shromažďovat jejich výsledků. Systém bude napsaný v programovacím jazyce Java a bude se skládat z dvou části, serverová část a klientská část.

Uživatel bude mít tento systém jako svůj řídicí, centrální a monitorovací nástroj, na serverové části bude moci definovat úkoly a posílat je po síti připojeným klientům (pracovním procesům).

úkoly budou formy:

- název aplikace, která musí být nainstalována na cílovém uzlu, aby ji klient mohl spouštět,
- seznam parametrů spouštění,
- parametr, který říká kolikrát má být daný úkol opakovaně spouštěn,
- soubor standardního výstupu aplikace, který se posílá zpátky k serveru.
- výstupní soubor aplikace: aplikace může mít vlastní výstupní soubor, proto je třeba ho zadat do úkolu (stačí cesta tam kde bude soubor).

Klienti zůstávají připojení k serveru dokud jsou ještě nějaké úkoly ke zpracování, a ukončují se ve chvíli, když dostanou od serveru flag, že už nejsou další úkoly na práci.

Uživatel bude mít možnost sledovat kolik klientů už je připravené na přijetí úkolů od serveru. A stisknutím tlačítka z GUI, začne server posílat úkoly klientům.

Všechny výstupní soubory, které byly nadefinovány v konfiguračním souboru, budou klienti posílat zpátky serveru, tím se splní jeden z hlavních účelů systému, že všechny výstupy se shromáždí na jednom místě (na serveru), pro snazší analýzu výsledků.

V konfiguračním souboru serveru kde jsou nadefinované úkoly, můžou být skupinové úkoly, tj. soubor jednotlivých úkolů, které musí být spouštěny současně, protože většinou potřebují si mezi sebou vyměnit data při běhu. Proto server začíná v posílání s těmi úkoly, a až skončí, začne posílat jednotlivé samostatné úkoly.

Systém bude mít dva konfigurační soubory, jeden hlavní, a druhé se bude možno editovat přímo v GUI.

Systém bude vytvořen jako dva jar balíky, jeden pro server a jeden pro klienta, s tím že u serveru bude GUI a u klienta bude stačit konzolová aplikace. Systém bude po překopírování spustitelný na jakémkoliv počítaci s Javou.

5.2 Případy užití

Systém bude hlavně ušetřovat uživatelovi čas, a usnadňovat mu práci s distribuovanými aplikacemi. A to díky hlavnímu konceptu systému, že všechno se bude řídit centrálně pomocí jednoho řídicího počítače (serveru).

Hlavní UML diagramy případů užití jsou uvedené pod přílohou C. Jednotlivé případy užití jsou popsány v následujících bodech:

- definovat úkoly:

  systém bude mít k dispozici hlavní konfigurační soubor, které bude obsahovat předem naplánované práce, které server bude posílat připojeným klientům. Uživatel bude tyto úkoly definovat podle definované struktury konfiguračního souboru (viz přílohu A). V případě, že uživatel udělá nějakou chybu, a v konfiguračním souboru bude špatně nadefinovaný úkol, systém, konkrétně klientská část systému bude hlásit, že úkol není validní, a pokračuje běh systému dál bez problému.

- Vytvořit nové úkoly:

  Uživatel bude moci přes GUI definovat, editovat, přidávat a mazat úkoly, které ale bude uložené v jiném konfiguračním souboru, než hlavní konfigurační soubor. Uživatel bude moci potom pomocí Swing FileChooser dialogu vybrat se kterým souborem bude systém pracovat.

- Opakovaně spouštět aplikace v distribuovaném systému a s různými vstupními parametry:

  Jak už bylo zmíněno ve specifikaci požadavků systému, u každého úkolu bude parametr, který určuje kolikrát má klient daný aplikaci spouštět a tu samou aplikaci může obsahovat více úkolů, s tím, že každý úkol bude mít jiné vstupní parametry aplikace.

- Analyzovat výstupů jednotlivých spouštění:

  Díky tomu, že klienti budou posílat výstupní soubory aplikací zpátky serveru, tak uživatel bude mít všechno, co potřebuje na analýzu výsledků na jednom místě.

5.3 Vstupy a výstupy systému

5.3.1 Vstup systému

Každá část systému bude mít jeden vstupní parametr, a to je konfigurační soubor pro klientskou aplikaci a jeden pro serverovou aplikaci. Konfigurační soubory se budou klasické Java `.property` souboy.

Konfigurační soubor serveru bude obsahovat:

- Port: číslo portu, na kterém bude server neustále poslouchat a přijímat požadavky o připojení od klientů.

- Definice jednotlivých úkolů a skupin úkolů.

Konfigurační soubor klienta bude jednodušší a bude obsahovat jenom číslo portu, kam má posílat požadavky o připojení a adresu, kde se nachází server na síti, ideálně IP adresu serveru.

Další vstup systému souvisí s jedním případem užití, a to je když bude chtít uživatel vytvořit nové úkoly, tak bude muset do GUI zadávat jednotlivé položky úkolu. (viz uživatelský manuál).

5.3.2 Výstup systému

Výstupem systému budou výstupy a výstupní soubory spouštěných aplikací, které se budou shromažďovat na serveru.

5.4 Použité technologie

Dvě hlavní technologie, které byly používané při implementaci systému jsou popsané v následujících podkapitolách.

5.4.1 Síťová komunikace

Na síťovou komunikaci mezi serverem a klientem je třeba si zvolit nejvhodnější technologii, která bude programátorovi stačit na vytvoření systému tak, aby byla splněna specifikace požadavků s nejmenšími náklady.

V Javě existuje několik technologií na síťovou komunikaci (viz 4. kapitolu). Pro účely práce by stačilo používat TCP/IP, vzhledem k rychlosti přenosu dat po síti, a k nízké režii na systém.

5.4.2 Grafické uživatelské rozhraní

Serverová část systému se bude ovládat přes grafické uživatelské rozhraní se standardním vzhledem. V Javě je několik technologií, které umožňují implementovat UI, nejrozšířenější technologie je Swing, proto se bude hodit pro implementaci systému.

Swing je dostupným základním frameworkem v Javě Core API, je velmi komplexní a má přes 30 komponent, všechny komponenty se vykreslují samy, a jejich vzhled je nezávislý na OS, což reprezentuje hlavní výhodu použití Swingu [25].

5.5 Struktura aplikace

Aplikace má síťovou architekturu, která odděluje klienta a server, kteří spolu komunikují přes počítačovou síť. Struktury jednotlivých částí aplikace jsou popsány v následujících podkapitolách.

5.5.1 Struktura aplikace serveru

Aplikace serveru bude mít třívrstvou architekturu, jednotlivé vrstvy jsou popsány v následujících bodech:

- Datová vrstva: server nebude mít žádnou svou klasickou databázi. Jediné data, se kterým bude pracovat jsou vstupní a výstupní data, kterými jsou textové soubory ze standardního výstupu či výstupu aplikace.
- Aplikační vrstva: aplikační vrstva se bude starat o celý chod serveru. Spouštění serveru, přijímání požadavků o připojení od klientů, posílání úkolů jednotlivým klientům, a na konec přijímání výstupních souborů, které klienti budou serveru posílat zpátky po dokončení výpočtu.
- Prezentační vrstva: Relativně jednoduchý grafické uživatelské rozhraní (GUI), přes které se bude systém ovládat. GUI bude napsané ve Swingu a bude mít standardní vzhled.

5.5.2 Struktura aplikace klienta

Aplikace klienta nebude mít grafické uživatelské rozhraní, bude stačit jako konzolová aplikace, protože hlavně bude sloužit k výpočtům. Jednotlivé vrstvy jsou popsány v následujících bodech:

- Datová vrstva: podobně jako u serveru. Pracovní data budou soubory výstupů a výstupů spouštěných aplikací.
- Aplikační vrstva: aplikační vrstva se bude starat o celý chod klienta. Spouštění klienta, posílání požadavků o připojení k serveru, přijímání úkolů na zpracování od serveru, a na konec posílání výstupních souborů zpátky k serveru po dokončení výpočtu.

6 Implementace

V následujících podkapitolách bude popsáno, jak byl systém implementován, jaké jsou jeho hlavní funkcionality a seznam hlavních tříd u každé z nich.

Systém byl implementován využitím standardních abstraktních datových typů Javy.

6.1 Třídy systému podle funkcionalit

V následujících bodech budou popsány hlavní funkcionality systému a jejich hlavní třídy.

6.1.1 Aplikační vrstva (příprava řídicího procesu)

Jde o spouštění hlavního procesu systému (spouštění serveru), který bude řídit ostatní pracovní procesy (klienty), a načítání příslušného konfiguračního souboru, kde jsou nadefinovány jednotlivé úkoly ke zpracování.

A následně přijímat požadavky o připojení od klientů a zahájení spolupráci. Spoluprací myslíme obousměrnou komunikaci server ↔ klient, kde server posílá úkoly a klienti posílají zpátky výstupy.

Jak už bylo zmíněno ve specifikaci požadavků, úkoly můžou být nadefinované jednotlivě a nebo ve skupině úkolů, které by měly běžet současně v distribuovaném prostředí, protože většinou probíhá mezi nimi výměna dat, nebudou-li tyto úkoly běžet současně, mohlo by dojít k zablokování nějakého procesu, který bude čekat na nějaký vstup od jiného procesu, se kterým měl běžet současně.

Proto server se chová tak, aby pravděpodobnost, že nějaký proces bude čekat dlouho na vstup od jiného procesu, aby byla co nejmenší. A to tak, že se podívá na konfigurační soubor a zjistí jestli jsou tam skupiny úkolů nadefinované a začne je posílat jako první, až když už bude všechny úkoly skupin vypracovány, pokračuje v posílání s jednotlivými úkoly.

Hlavní třídy:

- `MultiServer`: hlavní vlákno serveru, které načítá konfigurační soubor a inicializuje všechny objekty, potřebné pro další práci, jako seznamy jednotlivých i skupinových úkolů. Konfigurační soubor se přidává jako parametr z GUI, pomocí fileChooser dialog ve Swingu.

  Jakmile je všechno připravené na příjem požadavků připojení od klientů, otevře si server hlavní socket na hlavním portu, který byl nadefinován v konfiguračním souboru a začne poslouchat v cyklu a přijímat požadavky klientů.

  MultiServer třída má k dispozici objekt ArrayList<Socket>, který slouží k ukládání všechny nově vytvořené sockety pro připojené klienty. Výstupy všech přijatých požadavků se budou ukládat do toho ArrayListu, dokud nestiskne uživatel tlačítko „run jobs“ v GUI. Do této doby ještě nebyl poslán žádný úkol žádnému pracovnímu procesu (klientovi).

  Proces posílání úkolů připojeným klientům začne ve chvíli kdy se stiskne tlačítko „run jobs“. Server začne posílat, podle již zmíněného postupu, klinetům, jejichž sockety byly uložené do ArrayListu. Potom pro každý nový přicházející požadavek už se nebude nic ukládat do ArrayList<Socket> seznamu, ale rovnou se bude posílat úkoly, jsou-li ještě nějaké ke zpracování.

  Když už nebudou žádné úkoly ke zpracování, server odpovídá na požadavky klienty zprávou „terminate“, jako signál, že už nic nemá a že se mají ukončit.

  Toto hlavní vlákno zapisuje a aktualizuje statistiky v GUI, jako například, kolik máme připojených pracovních procesů (klientů), kolik už byly zpracováno úkolů, a tak podobně. A na konci vypíše „Finish“, jako že už se nic neposílá ze serveru, neboť už byly všechny úkoly zpracovány.

- `MultiServerThread`: na zpracování každého požadavku od klienta, hlavní vlákno serveru vytváří vlákno:

```
new MultiServerThread(serverSocket.accept(), ...)
                    .start();
```

které bude sloužit ke komunikaci s daným klientem, tj. Posílání úkolů a příjem výstupních souborů.

Toto vlákno bude také aktualizovat GUI, konkrétně JTable komponentu, kde budou zobrazené informace o připojených klientů (viz uživatelský manuál).

- `Clienta`: pracovní proces, který se musí spustit po spouštění serveru. Načítá ze svého konfiguračního souboru port a IP adresu serveru, a v cyklu začne posílat požadavky o připojení k serveru.

  Klient zůstává připojen, dokud ještě jsou nějaké úkoly na práci, jinými slovy, dokud nedostane od serveru signál „terminate“, jakmile ho dostane, ukončí se.

  Úkoly se posílají jako jeden řetězec, který potom musí klient parsovat a ověřovat, jestli jsou validní úkoly či ne, podle toho jaký je formát definice úkolu v konfiguračním souboru. Formát byl mnou vymyšlený, tak aby implementace byla co nejméně časově náročná (viz přílohu A).

  Každý ověřený úkol, bude dál posílán workeru (další pracovní proces pod klientem):

  ```
  Worker w = new Worker();
  runResult = w.runApp(app, params);
  ```

  , a po úspěšném dokončení úkolu, klient posílá výstupní soubor/y serverovi.

- `Worker`: tato třída přivezme úkol od klienta a vykonává ho. Jako atributy má dva, název aplikace (*.jar, *.exe, *.bat, ...), a druhý atribut je řetězec parametrů aplikace („par1, par2, par3, ...), který worker musí opravit do tvaru „par1 par2 par3 ...“, aby odpovídalo klasickému formátu předávání parametrů aplikaci na konzolovém řádku.

  Worker na spouštění aplikací používá objekt *ProcessBuilder*.

  Třída ProcessBuilder se používá na vytváření systémových procesů v Javě.

Po úspěšnému běhu aplikace, worker ukládá výstupní soubor aplikace do příslušného adresáře u klienta. Výstupní adresář obsahuje všechny výstupní soubory, které mají v názvu časovou značku, aby se jejich obsahy nepřepisovaly navzájem, a klient serveru pošle vždy posledně uložený výstupní soubor.

Obsah výstupní adresáře se maže po spouštění klienta, tedy klient vždycky má na začátku prázdnou složku, kam se bude ukládat výstupní soubory.

6.1.2 Aplikační vrstva (správa konfiguračního souboru)

Jde o možnost vytváření nových úkolů a jejich uložení do druhého konfiguračního souboru systému. Přímo z GUI bude moci uživatel definovat nové úkoly (viz uživatelský manuál), úkoly z konfiguračního souboru budou zobrazené v tabulce, a bude je možné editovat, mazat a přidávat nové. A na konec ukládat.

Před ukládáním se ověřuje, zda všechny položky jednotlivých úkolů jsou validní, například zda port je opravdu číslo a není řetězec.

- `DataTaskConfigFile`: tato třída má dvě hlavní metody, které jsou volány jako reakci na GUI události. První událost je vytvoření nového úkolu, zobrazí se Swing dialog, v jehož JTable budou řádky ze druhého konfiguračního souboru systému. Tedy, metoda `readConfigFile(String fileName)`, načítá obsah konfiguračního souboru a aktualizuje datový model Jtable dialogu.

  Druhá metoda `saveConfigFile(String fileName)`, ukládá obsah JTable tabulky do samého konfiguračního souboru. Před ukládáním.

6.1.3 Prezentační vrstva

Implementace prezentační vrstvy se týká jenom aplikace serveru. Implementace GUI ve Swing má největší počet tříd. Hlavní třídy jsou popsány v následujících bodech:

- `MainGUI`: tato třída vytvoří hlavní Jframe GUI systému, hlavní panel Jframe má BorderLayout rozložení, obsahuje jednotlivé záložky systému, „Main“, kde se hlavně zobrazuje obsah konfiguračního souboru serveru a „Slaves“ záložku, kde se

mimo jiné vypisuje statistiky a kde je hlavní Jtabulka obsahující detaily jednotlivých připojených klientů. (viz uživatelský manuál v příloze).

Tato třída obsahuje taky metodu `eventsService()` na obsluhu jednotlivých událostí systému.

- `AlFileChooserDialog`: třída na obsluhu události načítání konfiguračního souboru. Zobrazí se Swing JfileChooser, kde uživatel naviguje ve file systému OS a najde si příslušný konfigurační soubor, jehož obsah bude potom zobrazený pod záložkou „Main“.

- `AddNewTaskDialog`: tato třída je na obsluhu události definování nových úkolů a editaci stávajících úkolu ve druhém konfiguračním souboru systému.

  Dialog obsahuje hlavně JTable, kde řádky budou jednotlivé úkoly z konfiguračního souboru.

  Jsou ještě další třídy, související se GUI, jako například `DataClients`, `DataTask`, ty jsou datové třídy, obsahují statické struktury jednotlivých používaných Jtabulek v GUI. Třídy `DataModelTask`, `DataModelClients`, tyto třídy dědí od `AbstractTableModel` a jsou přiřazené příslušným Jtabulkám. Další třídy jsou už doplňující třídy jako třídy na řízení události z GUI.

7 Testování

Při testování aplikace jsem definoval dva konfigurační soubory, každý z nich má 20 úkolů, 14 samostatných a 6 skupinových úkolů. V prvním souboru (viz přílohu D.1), parametr, určující počet opakování spouštění úkolu má hodnotu jedna, ve druhém (viz přílohu D.2) má hodnotu dva.

Na testování jsem používal dvě vlastní aplikace, jedna se standardním výstupem, jehož velikost je v rozmezí stovek KB a druhá aplikace má výstup do souboru, jehož velikost je v rozmezí desítek MB.

K dispozici jsem měl tři notebooky, jeden hlavní 4-jádrový (A: 2.60 GHz – 4 GB RAM), a dva 2-jadrové, jeden (B: 2.10 GHz – 7.85 GM RAM) a druhý (C: 2.00 GHz – 1.00 GB RAM).

7.1 Testovací scénáře

Testovací scénáře jsou nastavené podle tří faktorů:

- počet pracovních procesů, které jsem spouštěl k serveru:

  tj. kolik spouštěných klientů.

- velikost výstupů (tedy velikost přenesených dat po síti):

  podle volby konfiguračního souboru se liší velikost výstupů.

- podle počet opakování spouštění úkolů:

  jak už bylo zmíněno, v jeden z konfiguračních souboru každý úkol se má spouštět dvakrát.

7.2 Výsledky

Výsledky jednotlivých testů jsou uvedené v následujících podkapitolách. Výsledky jsou znázorněné v tabulkové i grafové podobě.

7.2.1 První test

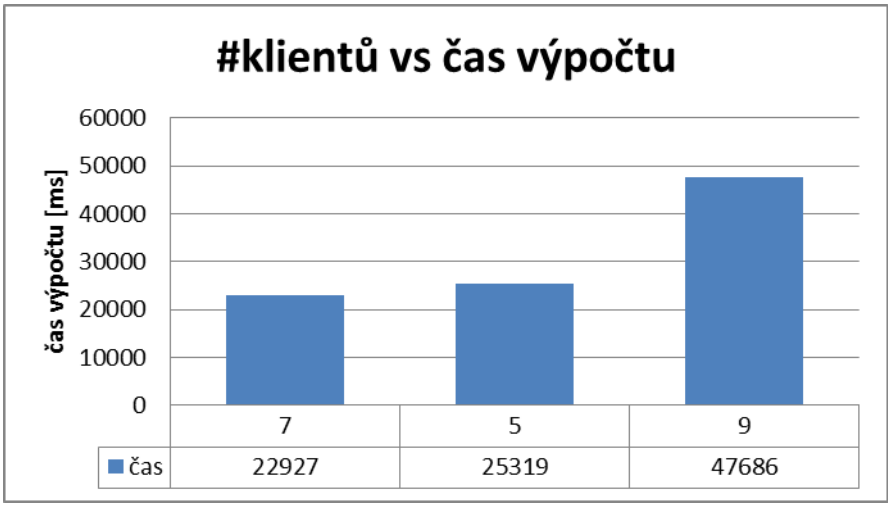
Na tento test jsem používal jenom dva ze tří notebooků. A a B notebooky. Server běžel na A a klienty a B a A (localhost). Oba konfigurační soubory byly používané. Výsledky jsou ukázané v tab. 4.

Poznámka: počet úkolů 18 * 2 znamená, že každý úkol měl běžet opakovaně dvakrát.

Tabulka 3 - výsledky použití 1. konfiguračního souboru

Počet prac. procesů	Počet úkolů	Objem přen. Dat [MB]	Čas výpočtu [ms]
7	**20**	**170**	**22927**
5	**20**	**170**	**25319**
9	**20**	**170**	**47686**

Graf výsledků pro lepší znázornění



Obrázek 9 - výsledky použizí 1. konfig souboru

Z grafu na obr. 9 je vidět, že ideální počet pracovních procesů pro takového testovacího případu (170 MB přenesených dat po bezdrátové síti s rychlostí cca 8 Mbps) je 7 pracovních procesů. S více procesy se čas přenosu zhoršuje kvůli režii vytváření a ukončování pracovních procesů.

A při použití druhého konfiguračního souboru, výsledky vypadaly následovně (viz tab. 4):

Tabulka 4 - výsledky použití 2. konfiguračního souboru

Počet prac. procesů	Počet úkolů	Objem přen. Dat [MB]	Čas výpočtu [ms]
7	**20 * 2**	**340**	**45057**
7	**20 * 2**	**340**	**46185**
5	**20 * 2**	**340**	**93812**
9	**20 * 2**	**340**	**91631**

Schválně jsem opakoval jeden běh se sedmi pracovními procesy, a vidíme, že čas přenosu není pokaždé stejný, což je normální, protože každý stroj má nějakou režii, i stav sítě není stabilní, probíhají jiné přenosy, které zatěžují síť.

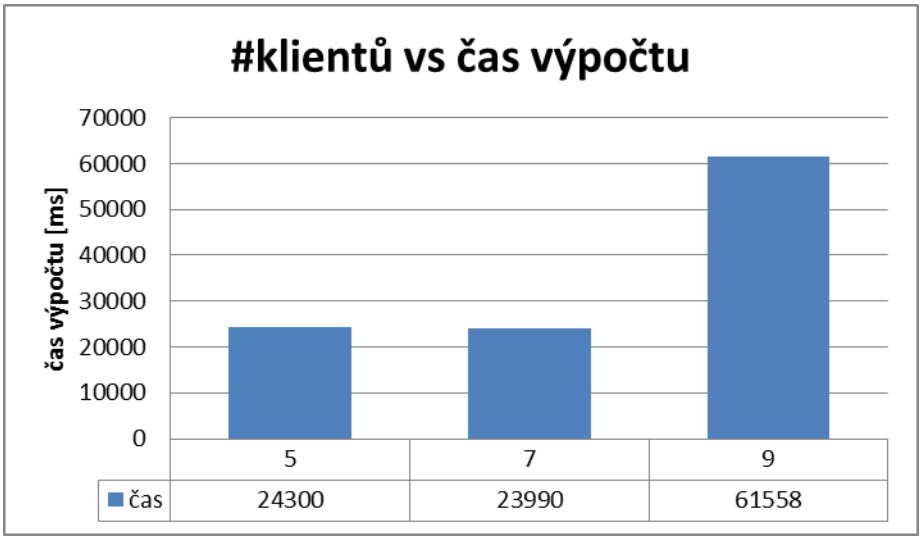
7.2.2 Druhý test

V tomto testu jsem postupoval stejně jako u předchozího testu, jediný rozdíl byl ve volbě serveru Tentokrát server běžel na B a klienty na A a B (localhost). Výsledky v tab. 5:

Tabulka 5 - výsledky použití 1. konf. Souboru

Počet prac. procesů	Počet úkolů	Objem přen. Dat [MB]	Čas výpočtu [ms]
7	**20**	**170**	**23990**
5	**20**	**170**	**24300**
9	**20**	**170**	**61558**

Pro lepší znázornění je i graf výsledku na obr



Obrázek 10 - výsledky použizí 2. konfig souboru

Tady také vidíme, že 7 pracovních procesů jsou ideální pro daný výpočet, zdůvodnění je stejné jako u předchozího testu.

Výsledky běhu s 2. Konfiguračním souborem jsou ukázány v tab 6.

Tabulka 6 - výsledky použití 2. konfig. Souboru

Počet prac. procesů	Počet úkolů	Objem přen. Dat [MB]	Čas výpočtu [ms]
7	**20 * 2**	**340**	**85082**
7	**20 * 2**	**340**	**88914**
9	**20 * 2**	**340**	**95772**

Všímáme si tady také rozdíl v čase přenosu se stejným počtem pracovních procesů, důvody jsou jasné z předchozího testu, také.

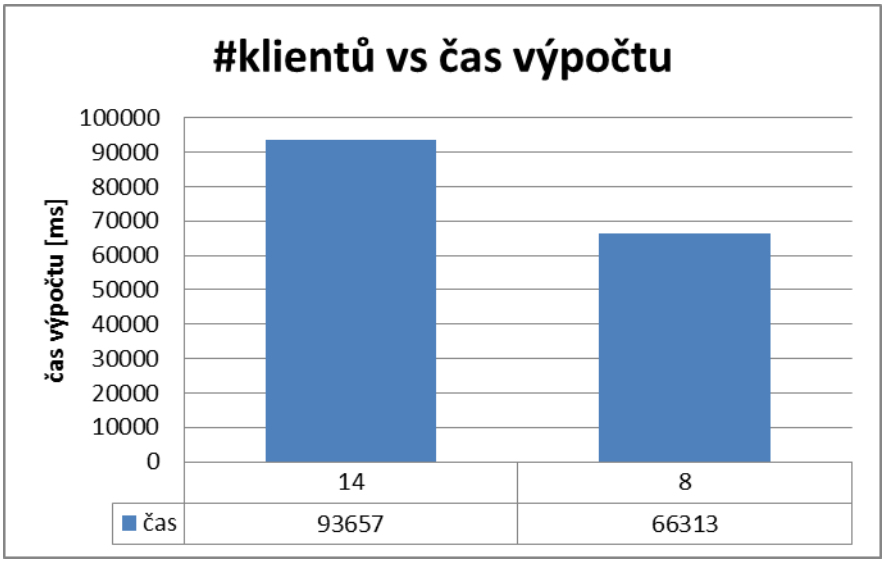
7.2.3 Třetí test

Tady u tohoto testu jsem používal třetí notebook C, sice je slabý stroj ve srovnání s ostatními dvěma, ale na testování sloužil docela dobře. První výsledky jsou zobrazené v následující tab 7.

Tabulka 7 - výsledky použití 1. konf. Souboru

Počet prac. procesů	Počet úkolů	Objem přen. Dat [MB]	Čas výpočtu [ms]
14	**20**	**170**	**93657**
8	**20**	**170**	**66313**

Pro lepší znázornění je k dispozici i graf na obr. 11.



Obrázek 11 - výsledky použití 1: konfig souboru

Vidíme, že tady máme horší čas přenosů. Důvod je zřejmý, protože tento čas zahrnuje celkovou dobu komunikace mezi serverem a klienty, tj. od chvíle když uživatel stiskne tlačítko „run jobs“ až po zpracování posledního úkolu.

Tedy, režie je docela velká v tomto případě, vytvářet a ukončovat vlákna pro každý požadavek připojení není levná záležitost.

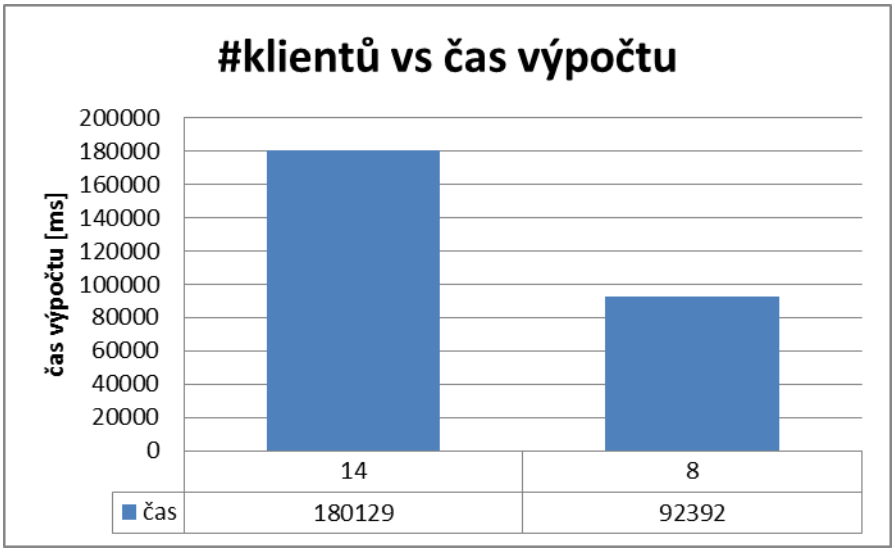
Výsledky použití 2. Konfiguračního souboru jsem zobrazená v následující tab. 8.

Tabulka 8 - výsledky použití 2. konfiguračního souboru

Počet prac. procesů	Počet úkolů	Objem přen. Dat [MB]	Čas výpočtu [ms]
14	**20 * 2**	**340**	**180129**
8	**20 * 2**	**340**	**92393**

Všímáme si tady taky, že čas přenosu je horší, důvody jsou jasné z předchozího testu.

Pro lepší znázornění je na obr. 12 graf výsledků s použití 2. Konfiguračního souboru.



Obrázek 12 - výsledky použití 2. konfig souboru

8 Závěr

Tématem této práce bylo vytvoření systému pro automatické spouštění existujících Java programů a shromáždění výsledků spouštění v distribuovaném prostředí. Ideálním příkladem použití systému bude například při simulaci silniční dopravy, kde uživatel bude moci spouštět simulační aplikace na několika počítačích v distribuovaném systému, pokaždé s jinými parametry a opakovaně kolikrát bude potřeba.

Analýza této práce byla rozdělena na dvě základní oblasti, první oblast byla o zkoumání existujících možností pro opakované spouštění programů v distribuovaném prostředí se zaměřením na jazyk Java, jako například JPVM a mpiJava. Druhá oblast byla o seznámení s možnostmi (vysokoúrovňové) síťové komunikace v jazyce Java. Důsledkem analýzy byl, že na systém se bude implementovat vlastní implementací s použitím TCP/IP pro síťovou komunikaci, zejména kvůli jeho rychlosti při přenosu dat.

Implementace systému byla rozdělena na dvě základní části, implementace serverové části systému, což byla třívrstová aplikace, a implementace klientské části systému, která měla jenom dvě vrstvy, aplikační a datovou, protože klientská aplikace slouží jenom k výpočtu, tedy nemusí mít grafické uživatelské rozhraní.

Systém byl otestován a všechny jeho funkčnosti fungovaly bez problému, tedy lze konstatovat, že práce splňuje zadání.

Literatura

[1] synchronizace v Javě,
http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html [9.6.2014]

[2] obrázek systému se sdílenou pamětím,
http://en.wikipedia.org/wiki/File:Shared_Memory.jpeg [9.6.2014]

[3] distribuované aplikace,
http://sofe2.pepiino.cz/wiki/doku.php?id=distribuovane_aplikace [10.6.2014]

[4] distribuované systémy,
http://csis.pace.edu/~marchese/CS865/Lectures/Chap1/Chapter1a.htm [10.6.2014]

[5] Remote Computing systems, http://dl.acm.org/citation.cfm?id=1464164 [10.6.2014]

[6] Cloud computing, http://priyathevaishnavite.wordpress.com/ [11.6.2014]

[7] Selenium, http://docs.seleniumhq.org/ [11.6.2014]

[8] PVM, http://www.csm.ornl.gov/pvm/ [11.6.2014]

[9] vlákna a procesy, http://www.cs.vsb.cz/benes/vyuka/pte/texty/vlakna/ch01s01.html [12.6.2014]

[10] PVM demon, http://is.muni.cz/th/72582/fi_b/PVMproDiVinE.txt [12.6.2014]

[11] JPVM, http://www.cs.virginia.edu/~ajf2j/jpvm.html [13.6.2014]

[12] MPI, http://www.mcs.anl.gov/research/projects/mpi/ [13.6.2014]

[13] mpiJava, http://surface.syr.edu/cgi/viewcontent.cgi?article=1006&context=npac [14.6.2014]

[14] MPI vs PVM, http://www-unix.mcs.anl.gov/ ~gropp/bib/papers/2002/mpiandpvm.pdf [14.6.2014]

[15] TCP/IP, http://searchnetworking.techtarget.com/definition/TCP-IP [17.6.2014]

[16] TCP/IP struktura, http://site.the.cz/index.php?id=3 [17.6.2014]

[17] TCP/IP struktura 2, drogo.fme.vutbr.cz/~jroupec/nsite/ppt/vpn-09.ppt [17.6.2014]

[18] JMS, http://docs.oracle.com/javaee/6/tutorial/doc/bncdr.html [18.6.2014]

[19] Java WebSocket, http://docs.oracle.com/javaee/7/tutorial/doc/websocket001.htm#BABDABHF [19.6.2014]

[20] RMI, http://docs.oracle.com/javase/tutorial/rmi/ [19.6.2014]

[21] RMI 2, http://www.cs.vsb.cz/grygarek/dosys/inprise/rmi/rmi.html [19.6.2014]

[22] RMI vs CORBA, http://www.oracle.com/technetwork/articles/javase/rmi-corba-136641.html [20.6.2014]

[23] CORBA, http://www.ece.rutgers.edu/~parashar/Classes/ece451-566/slides/lecture-dist-obj-corba-liu.pdf [20.6.2014]

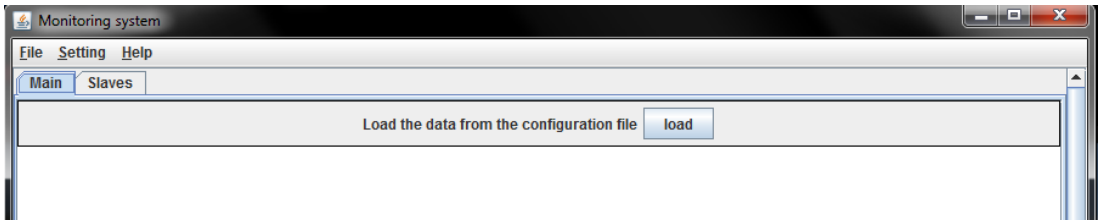
[24] RMI vs CORBA 2, http://www.javacoffeebreak.com/articles/rmi_corba/ [21.6.2014]

[25] Swing, http://docs.oracle.com/javase/tutorial/uiswing/ [22.6.2014]

Příloha A Uživatelský manuál

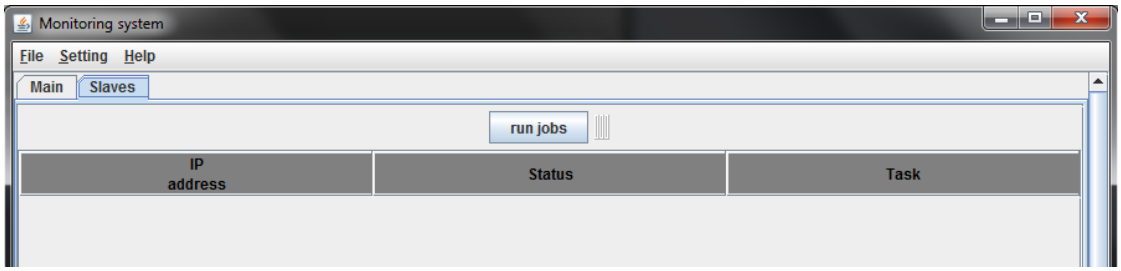
A.1 Popis GUI

GUI je jenom u serverové části systému. GUI je relativně jednoduché. Po spouštění serveru se uživateli zobrazuje hlavní okno systému (viz obr. 13)



Obrázek 13 - hlavní okno GUI serveru

jak je vidět na obrázku, GUI má dvě záložky „Main“ a „Slaves“. Pod záložkou „Main“ je tlačítko „load“ na načítání konfiguračního souboru systému, a pod tlačítko je textová plocha pro zobrazení obsahu konfiguračního souboru.



Obrázek 14 - záložka Slaves

Pod záložkou „Slaves“ (viz obr. 14) je tlačítko „run jobs“, po jeho stisknutí, začne server posílat úkoly připojeným pracovním procesům. Napravo od tlačítka „run jobs“ jsou tři textové plochy pro zobrazování statistik.

Hlavní obsah záložky je tabulka, která bude obsahovat informace o připojených klientech (IP adresa, status: connected či terminated a úkol, který byl klientovi poslán).

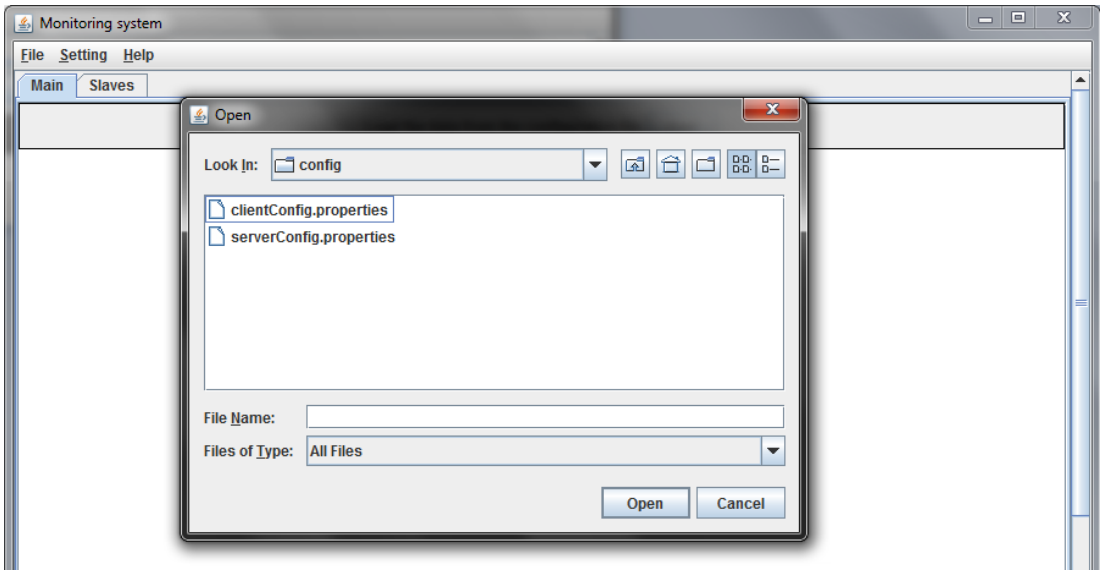
A.2 Funkce systému

Popisy jednotlivých funkcí systému jsou popsány v následujících podkapitolách.

A.2.1 Načítání konfiguračního souboru

Uživatel má dvě možnosti jak z GUI načítat konfigurační soubor systému, buď přímo ze záložky „Main“ tlačítkem „load“ a nebo z nabídky „File“ položkou „load config“.

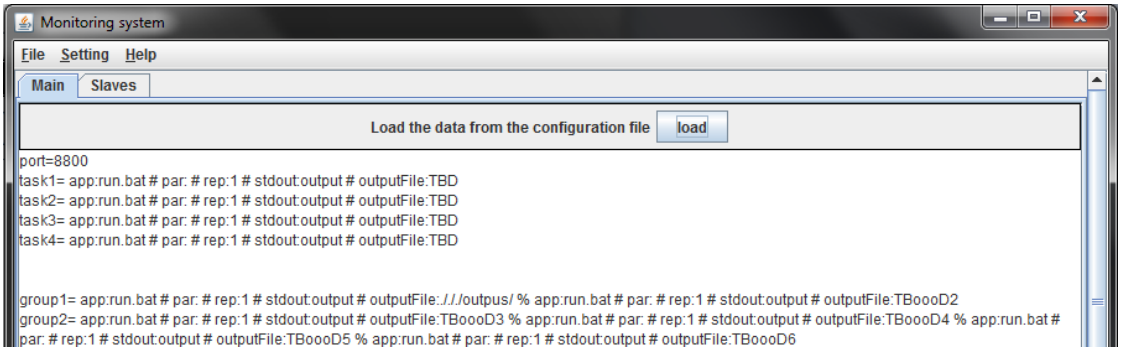
Po konání jedné z výše zmíněných možnosti se uživateli zobrazí JfileChooser (viz obr. 15), kde pak uživatel nejde příslušný konfigurační soubor a načítá jeho obsah do GUI.



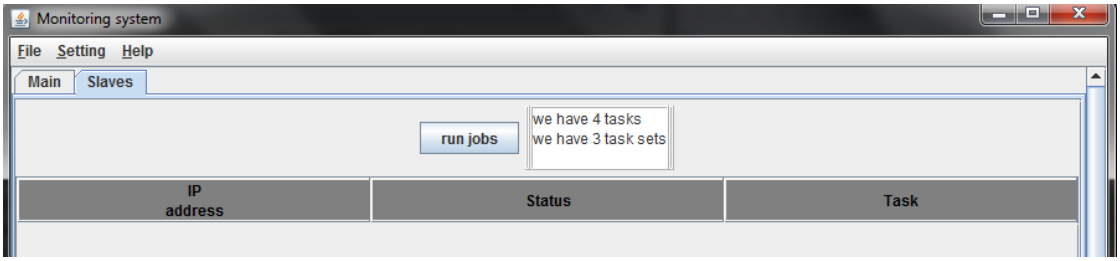
Obrázek 15 - načítání konfiguračního souboru

Uživatel potom uvidí obsah konfiguračního souboru pod záložkiu „Main“, a bude mít obecný přehled, jaké úkoly se bude konat na klientech (viz obr. 16).

Stručný přehled se objeví pod záložkou „slaves“, kde bude zobrazeno kolik je samostatných úkolů a kolik je skupinových v již načítaném konfiguračním souboru (viz obr. 17).



Obrázek 16 - předhled úkolů

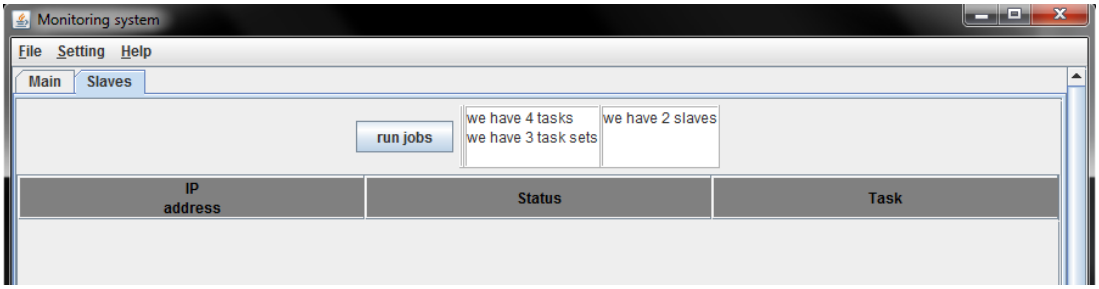


Obrázek 17 - předhled úkolůpod Slaves

A.2.2 Spouštění pracovních procesů

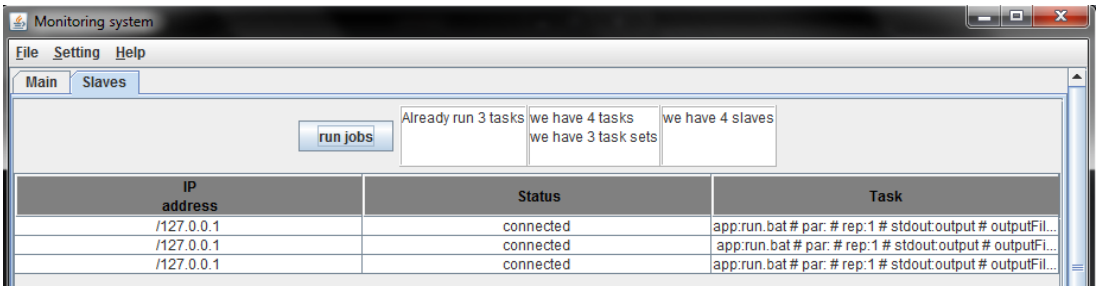
Ve chvíli kdy server už běží můžeme spustit klienty. Klienty se spouští pomocí dávkového souboru *rub.bat*, v tomto dávkovém souboru je nastavený parametr klientské aplikace, tj. konfigurační souboe *clientConfig.properties*.

Klient po jeho spouštění pošle požadavek o připojení k serverovi. Server požadavek přijímá a v GUI pod záložkou „slaves“ se objeví další textová plocha, v níž bude zobrazena zpráva o počtu přijatých klientů (viz obr. 18).



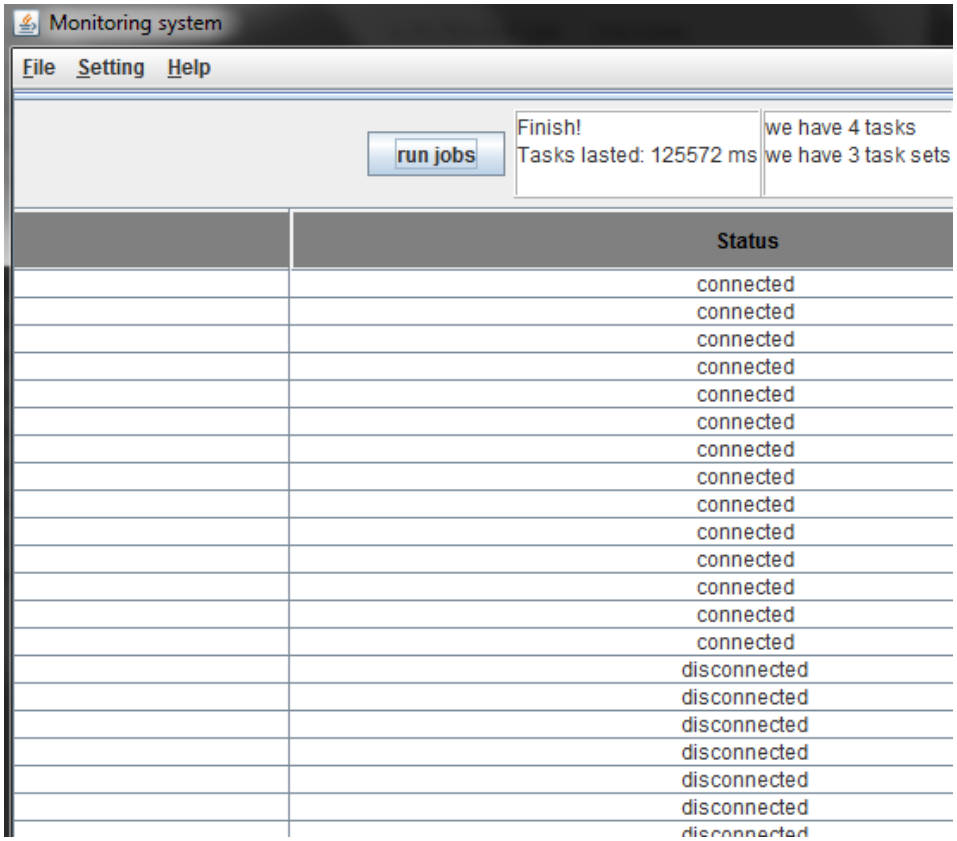
Obrázek 18 - počet připojených klientů

Uživatel, když uvidí, že už má dostatečný počet pracovních procesů na práci, stiskne tlašítko „run jobs“, tím se zahájí proces posílání úkolů jednotlivým klientům. V GUI se objeví třetí textová plocha, kde na začátku bude zobrazený aktuální počet zpracovaných úkolů a na konci bude zobrazena zpráva „finish“ s časem, jak dlouho trvalo zpracování všech úkolů (viz obr. 19).



Obrázek 19 - předhled běhu úkolů

V tabulce pod tím budou zobrazené detaily o připojených klientech (obr. 20).



Obrázek 20 - detaily pracovních procesů

Po skončení zpracování úkolů, všichni klienti se ukončí a na serveru v příslušných složkách bude uložené všechny výstupní soubory, které klienti posílali serveru postupně za běhu.

B Konfigurační soubory

1. Ukázka konfiguračního souboru serveru

   serverConfig.properties:

   odělovač mezi položkami úkolu je: #

   odělovač mezi položkami skupiny je: **%**

   port=8800 // port na kterém bude server neustále poslouchat

   task1= app:run.bat # par: par1, par2, par3 # rep:1 # stdout:StandardOutputFile # outputFile: appOutputFile

   task2= app:run.bat # par: par1, par2, par3 # rep:1 # stdout:StandardOutputFile # outputFile: appOutputFile

   group1= app:run.bat # par: par1, par2, par3 # rep:1 # stdout:StandardOutputFile # outputFile: appOutputFile % app:run.bat # par: par1, par2, par3 # rep:1 # stdout:StandardOutputFile # outputFile: appOutputFile

   group2= ...

2. Ukázka konfiguračního souboru klienta:

   clientConfig.properties:

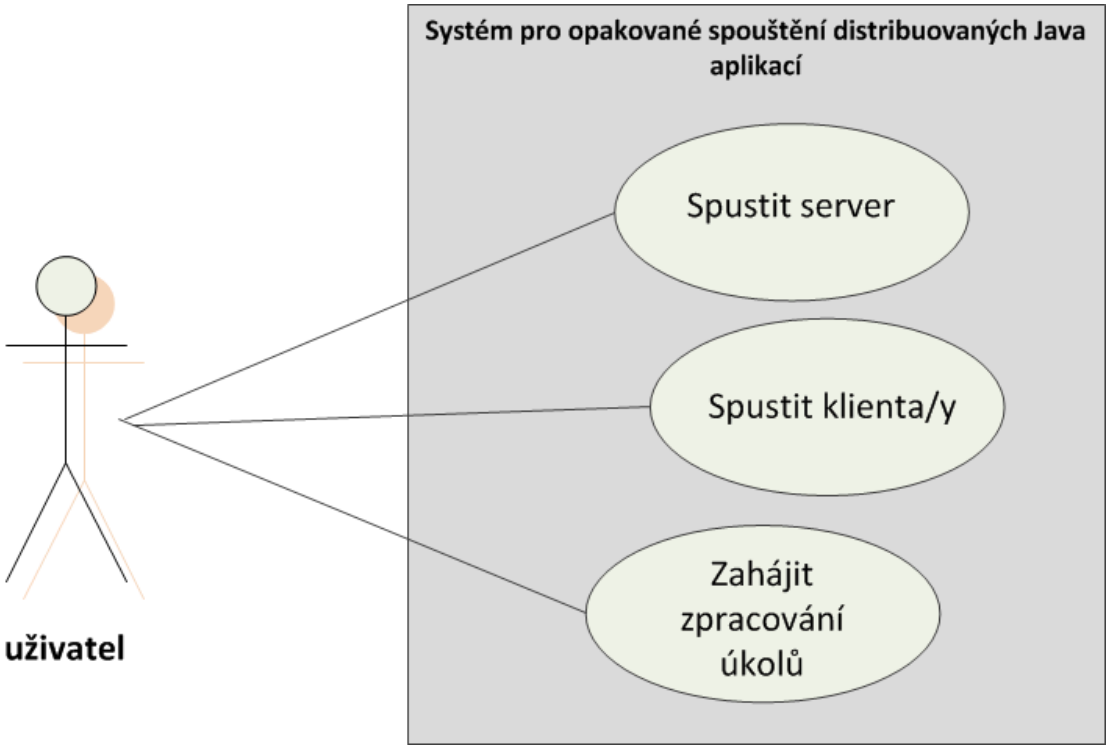
   server=localhost

   port=8800

C UML diagramy

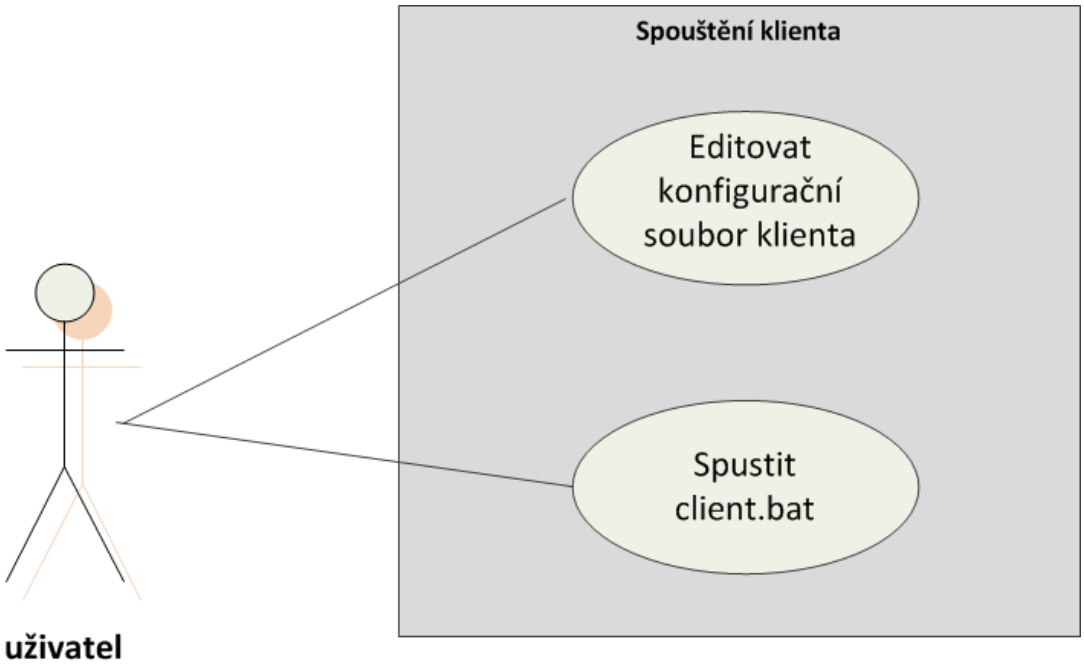
C.1 případy užití

C.1.1. obecné případy užití systému



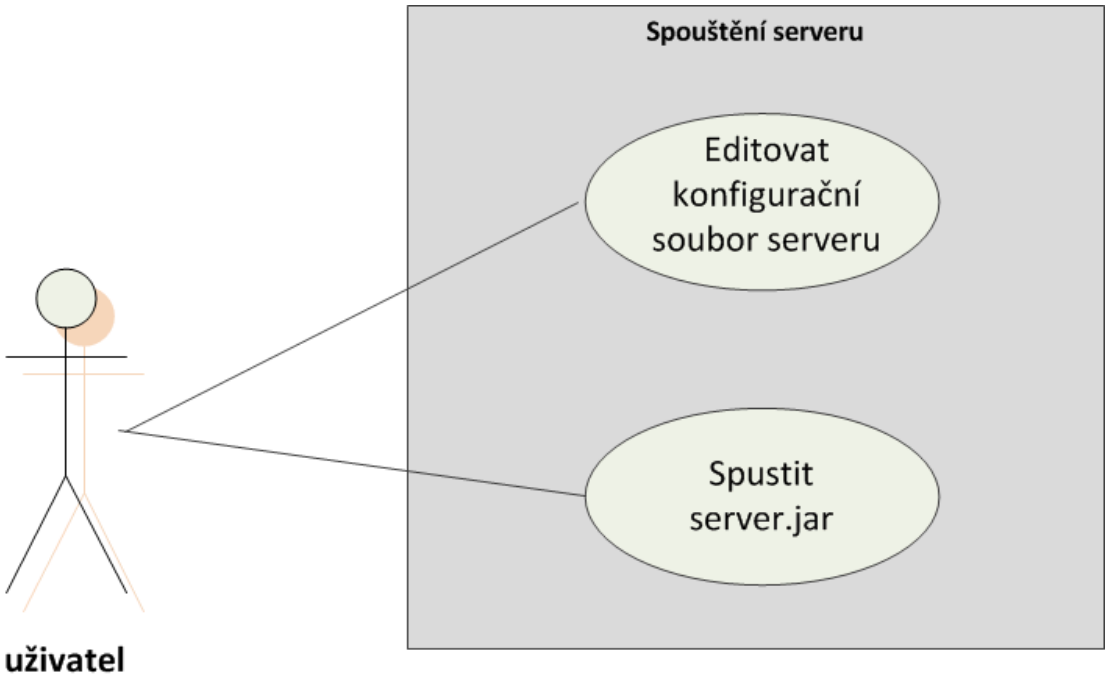
Obrázek 21 - obecný diagram případů užití

C.1.2 případ užití pro spouštění klienta



Obrázek 22 - případ užití spouštění klienta

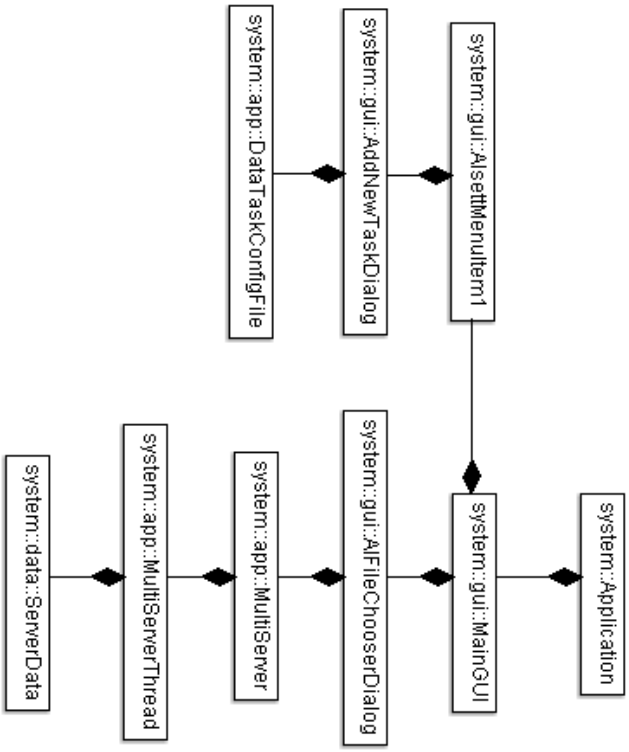
C.1.3 případ užití pro spouštění serveru

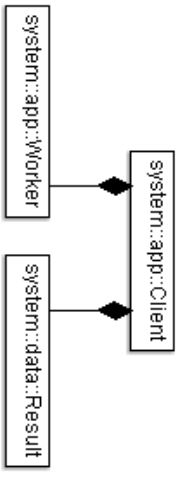


Obrázek 23 - případ užití spouštění serveru

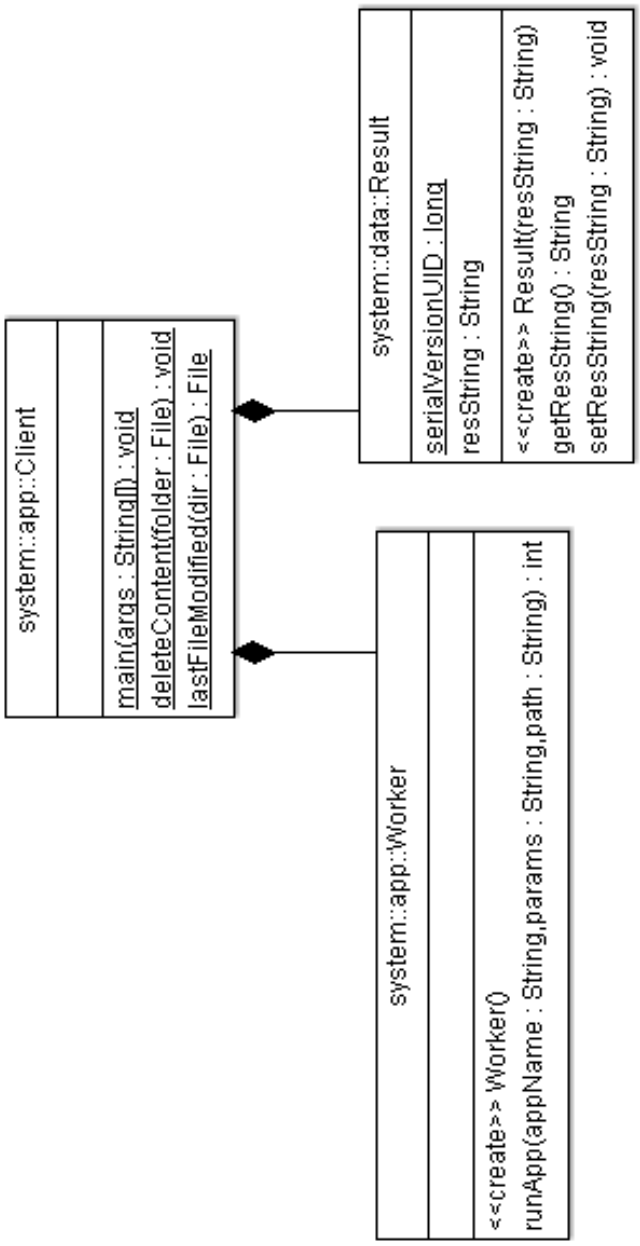
C.2 diagram tříd

C.2.1 Jednoduchý diagram hlavních tříd. Diagram tříd jednotlivých částí aplikace jsou uvedené zvlášť v následujících sekcích.

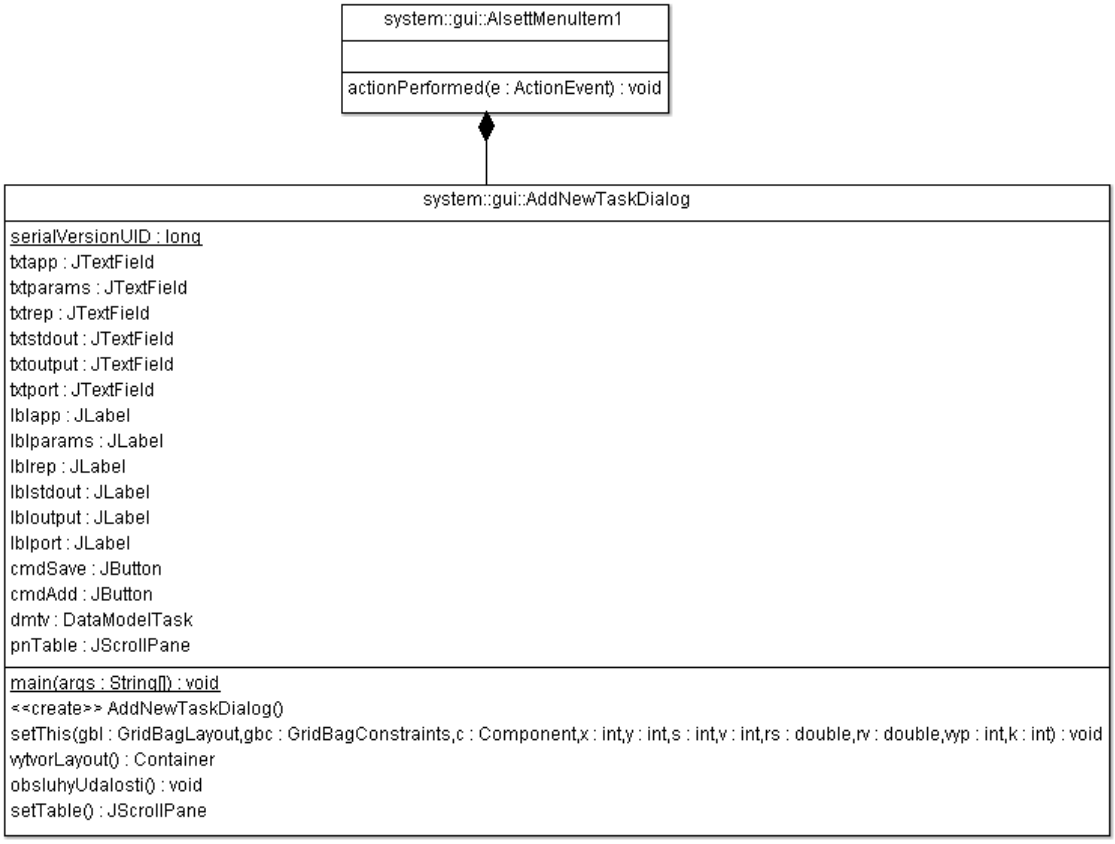




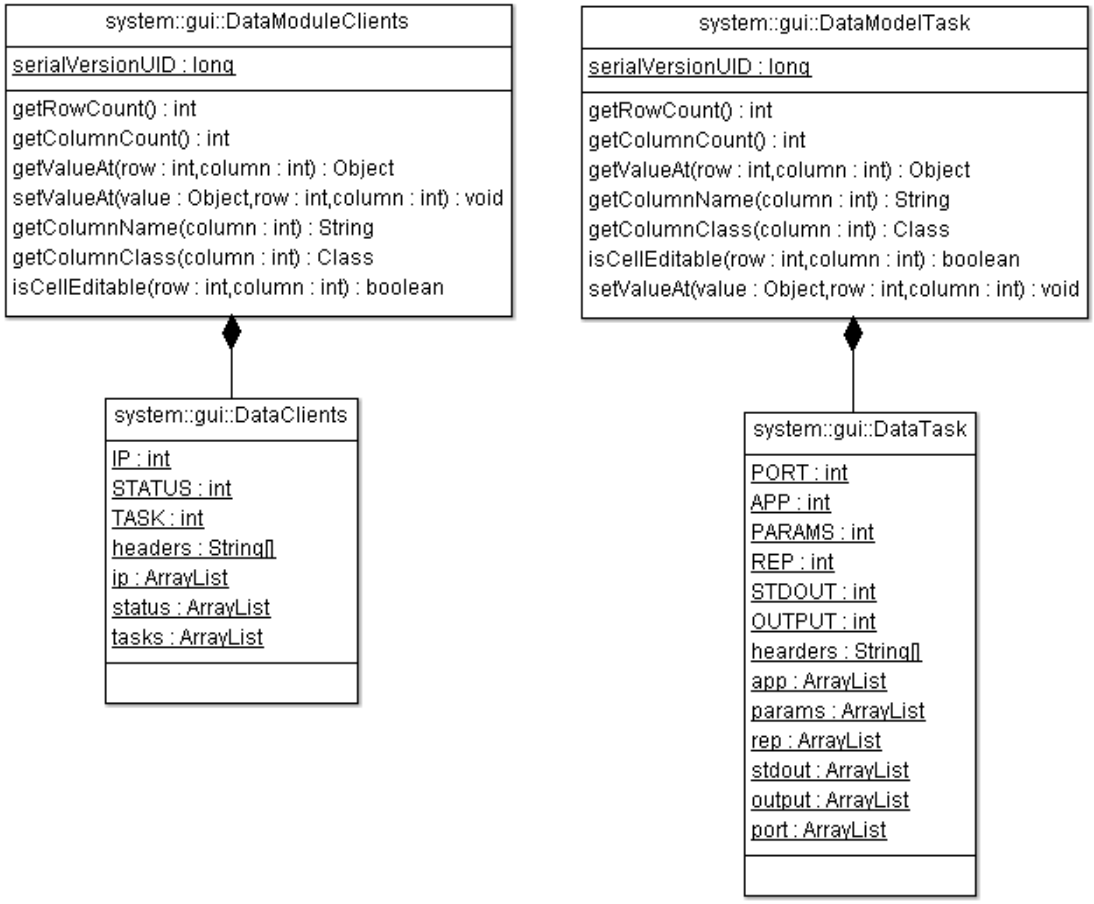
C.2.2


system::app::Client
main(args : String[]) : void
deleteContent(folder : File) : void
lastFileModified(dir : File) : File
system::data::Result
serialVersionUID : long
resString : String
<<create>> Result(resString : String)
getResString() : String
setResString(resString : String) : void
system::app::Worker
<<create>> Worker()
runApp(appName : String,params : String,path : String) : int

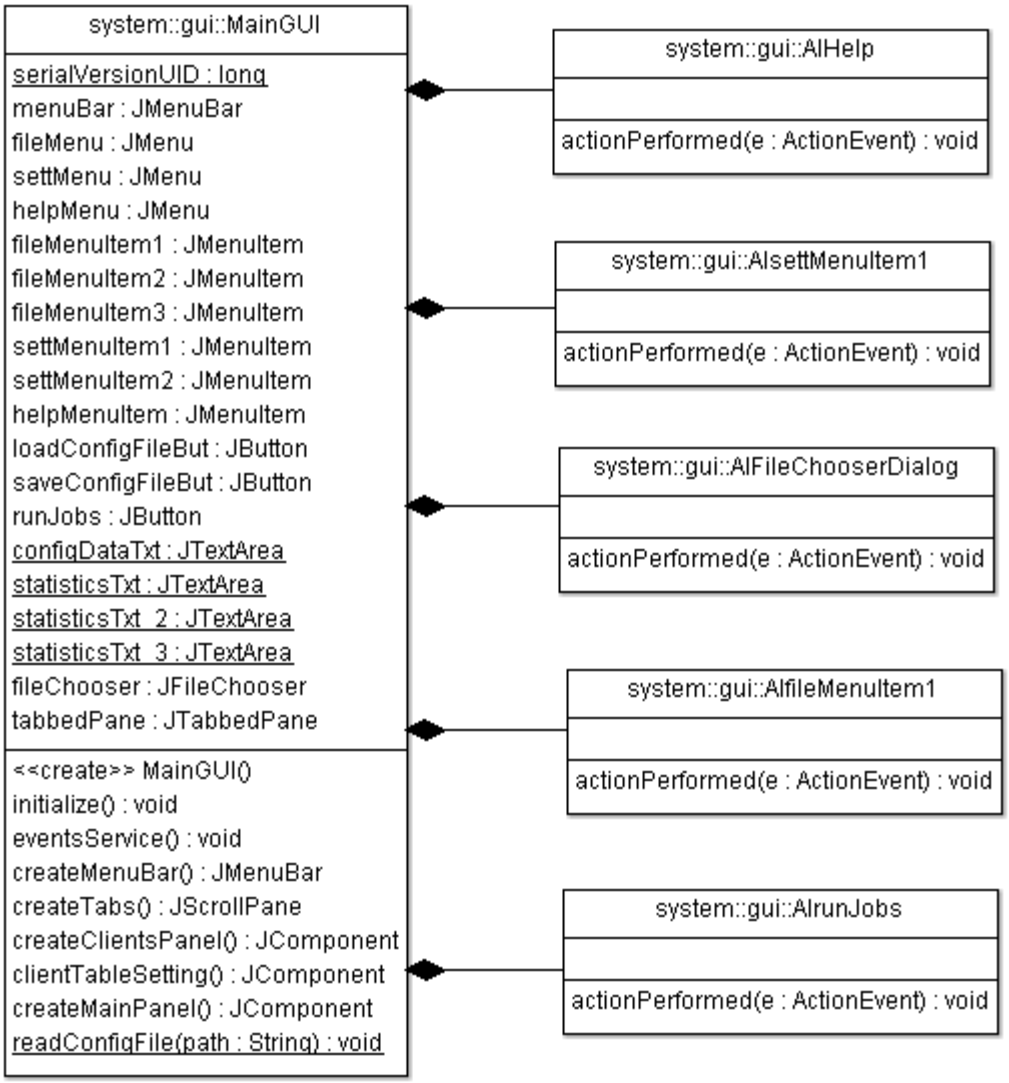
C.2.3


system::gui::AlsettMenuItem1
actionPerformed(e : ActionEvent) : void
system::gui::AddNewTaskDialog
serialVersionUID : long
txtapp : JTextField
txtparams : JTextField
txtrep : JTextField
txtstdout : JTextField
txtoutput : JTextField
txtport : JTextField
lblapp : JLabel
lblparams : JLabel
lblrep : JLabel
lblstdout : JLabel
lbloutput : JLabel
lblport : JLabel
cmdSave : JButton
cmdAdd : JButton
dmtv : DataModelTask
pnTable : JScrollPane
main(args : String[]) : void
<<create>> AddNewTaskDialog()
setThis(gbl : GridBagLayout,gbc : GridBagConstraints,c : Component,x : int,y : int,s : int,v : int,rs : double,rv : double,vyp : int,k : int) : void
vytvorLayout() : Container
obsluhyUdalosti() : void
setTable() : JScrollPane

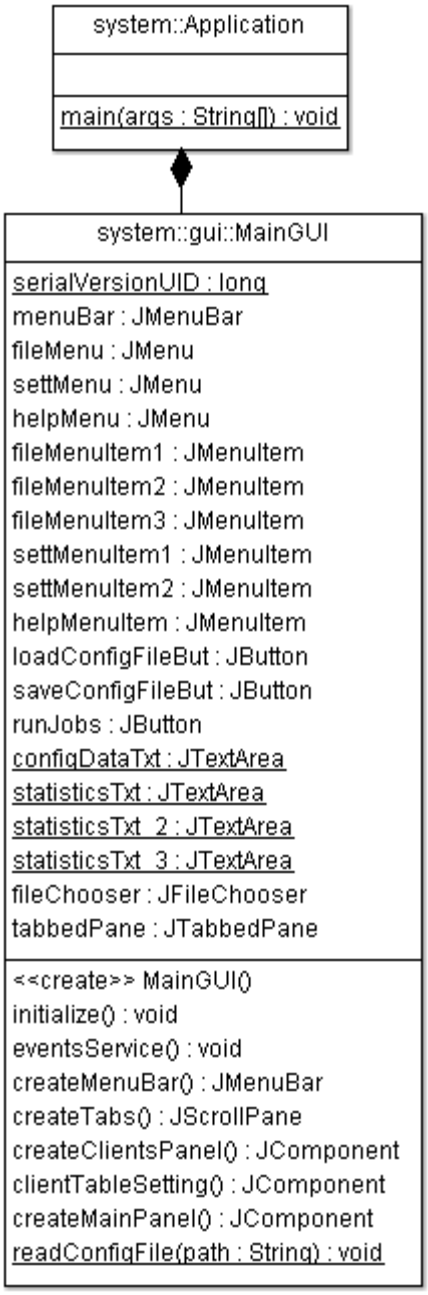
C.2.4


system::gui::DataModuleClients
serialVersionUID : long
getRowCount() : int
getColumnCount() : int
getValueAt(row : int,column : int) : Object
setValueAt(value : Object,row : int,column : int) : void
getColumnName(column : int) : String
getColumnClass(column : int) : Class
isCellEditable(row : int,column : int) : boolean
system::gui::DataClients
IP : int
STATUS : int
TASK : int
headers : String[]
ip : ArrayList
status : ArrayList
tasks : ArrayList
system::gui::DataModelTask
serialVersionUID : long
getRowCount() : int
getColumnCount() : int
getValueAt(row : int,column : int) : Object
getColumnName(column : int) : String
getColumnClass(column : int) : Class
isCellEditable(row : int,column : int) : boolean
setValueAt(value : Object,row : int,column : int) : void
system::gui::DataTask
PORT : int
APP : int
PARAMS : int
REP : int
STDOUT : int
OUTPUT : int
hearders : String[]
app : ArrayList
params : ArrayList
rep : ArrayList
stdout : ArrayList
output : ArrayList
port : ArrayList

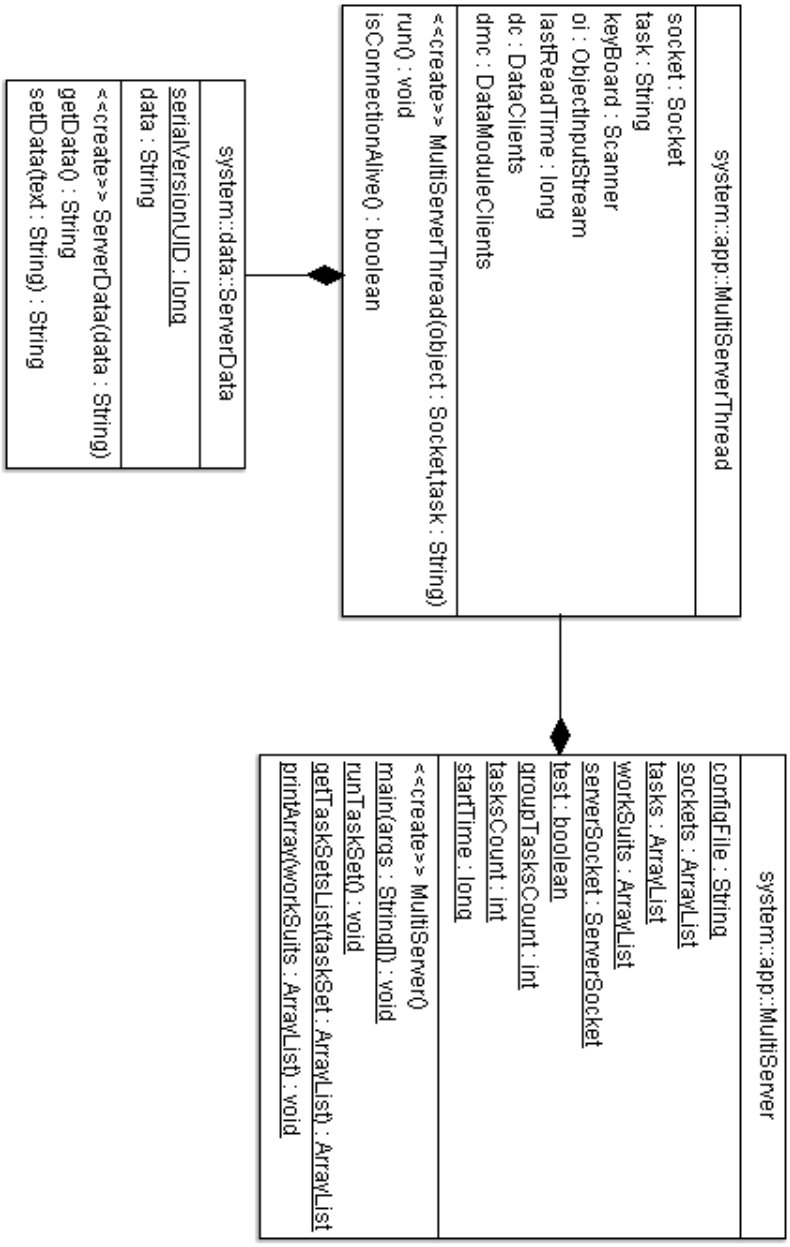
C.2.5


system::gui::MainGUI
serialVersionUID : long
menuBar : JMenuBar
fileMenu : JMenu
settMenu : JMenu
helpMenu : JMenu
fileMenuItem1 : JMenuItem
fileMenuItem2 : JMenuItem
fileMenuItem3 : JMenuItem
settMenuItem1 : JMenuItem
settMenuItem2 : JMenuItem
helpMenuItem : JMenuItem
loadConfigFileBut : JButton
saveConfigFileBut : JButton
runJobs : JButton
configDataTxt : JTextArea
statisticsTxt : JTextArea
statisticsTxt_2 : JTextArea
statisticsTxt_3 : JTextArea
fileChooser : JFileChooser
tabbedPane : JTabbedPane
<<create>> MainGUI()
initialize() : void
eventsService() : void
createMenuBar() : JMenuBar
createTabs() : JScrollPane
createClientsPanel() : JComponent
clientTableSetting() : JComponent
createMainPanel() : JComponent
readConfigFile(path : String) : void
system::gui::AlHelp
actionPerformed(e : ActionEvent) : void
system::gui::AlsettMenuItem1
actionPerformed(e : ActionEvent) : void
system::gui::AlFileChooserDialog
actionPerformed(e : ActionEvent) : void
system::gui::AlfileMenuItem1
actionPerformed(e : ActionEvent) : void
system::gui::AlrunJobs
actionPerformed(e : ActionEvent) : void

C.2.6


system::Application
main(args : String[]) : void
system::gui::MainGUI
serialVersionUID : long
menuBar : JMenuBar
fileMenu : JMenu
settMenu : JMenu
helpMenu : JMenu
fileMenuItem1 : JMenuItem
fileMenuItem2 : JMenuItem
fileMenuItem3 : JMenuItem
settMenuItem1 : JMenuItem
settMenuItem2 : JMenuItem
helpMenuItem : JMenuItem
loadConfigFileBut : JButton
saveConfigFileBut : JButton
runJobs : JButton
configDataTxt : JTextArea
statisticsTxt : JTextArea
statisticsTxt_2 : JTextArea
statisticsTxt_3 : JTextArea
fileChooser : JFileChooser
tabbedPane : JTabbedPane
<<create>> MainGUI()
initialize() : void
eventsService() : void
createMenuBar() : JMenuBar
createTabs() : JScrollPane
createClientsPanel() : JComponent
clientTableSetting() : JComponent
createMainPanel() : JComponent
readConfigFile(path : String) : void

C.2.7


system::app::MultiServerThread
socket : Socket
task : String
keyBoard : Scanner
oi : ObjectInputStream
lastReadTime : long
dc : DataClients
dmc : DataModuleClients
<<create>> MultiServerThread(object : Socket,task : String)
run() : void
isConnectionAlive() : boolean
system::data::ServerData
serialVersionUID : long
data : String
<<create>> ServerData(data : String)
getData() : String
setData(text : String) : String
system::app::MultiServer
configFile : String
sockets : ArrayList
tasks : ArrayList
workSuits : ArrayList
serverSocket : ServerSocket
test : boolean
groupTasksCount : int
tasksCount : int
startTime : long
<<create>> MultiServer()
main(args : String[]) : void
runTaskSet() : void
getTaskSetsList(taskSet : ArrayList) : ArrayList
printArray(workSuits : ArrayList) : void

D Konfigurační soubory testů

D.1

port=8004

task1= app:run.bat # par: 1000 Gauss -test # rep:1 # stdout:output # outputFile:

task2= app:run.bat # par:1000 Gauss -test # rep:1 # stdout:output # outputFile:

task3= app:run.bat # par:1000 Gauss -test # rep:1 # stdout:output # outputFile:

task4= app:run.bat # par: 1000 Gauss -test # rep:1 # stdout:output # outputFile:

task5= app:run.bat # par: # rep:1 # stdout:output # outputFile:

task6= app:run.bat # par: # rep:1 # stdout:output # outputFile:

task7= app:run.bat # par: # rep:1 # stdout:output # outputFile:

task8= app:run.bat # par: # rep:1 # stdout:output # outputFile:

task10= app:fileGen.jar # par: testFolder # rep:1 # stdout:output # outputFile: testFolder

task12= app:fileGen.jar # par: testFolder # rep:1 # stdout:output # outputFile: testFolder

task13= app:fileGen.jar # par: testFolder # rep:1 # stdout:output # outputFile: testFolder

task14= app:fileGen.jar # par: testFolder # rep:1 # stdout:output # outputFile: testFolder

group1= app:run.bat # par:1000 Gauss -test # rep:1 # stdout:output # outputFile: % app:run.bat # par: 1000 Gauss -test # rep:1 # stdout:output # outputFile: % app:run.bat # par: 1000 Gauss -test # rep:1 # stdout:output # outputFile:

group2= app:fileGen.jar # par: testFolder # rep:1 # stdout:output # outputFile: testFolder % app:fileGen.jar # par: testFolder # rep:1 # stdout:output # outputFile: testFolder % app:fileGen.jar # par: testFolder # rep:1 # stdout:output # outputFile: testFolder

D.2

port=8004

task1= app:run.bat # par: 1000 Gauss -test # rep:2 # stdout:output # outputFile:

task2= app:run.bat # par:1000 Gauss -test # rep:2 # stdout:output # outputFile:

task3= app:run.bat # par:1000 Gauss -test # rep:2 # stdout:output # outputFile:

task4= app:run.bat # par: 1000 Gauss -test # rep:2 # stdout:output # outputFile:

task5= app:run.bat # par: # rep:2 # stdout:output # outputFile:

task6= app:run.bat # par: # rep:2 # stdout:output # outputFile:

task7= app:run.bat # par: # rep:2 # stdout:output # outputFile:

task8= app:run.bat # par: # rep:2 # stdout:output # outputFile:

task10= app:fileGen.jar # par: testFolder # rep:2 # stdout:output # outputFile: testFolder

task12= app:fileGen.jar # par: testFolder # rep:2 # stdout:output # outputFile: testFolder

task13= app:fileGen.jar # par: testFolder # rep:2 # stdout:output # outputFile: testFolder

task14= app:fileGen.jar # par: testFolder # rep:2 # stdout:output # outputFile: testFolder

group1= app:run.bat # par:1000 Gauss -test # rep:2 # stdout:output # outputFile: % app:run.bat # par: 1000 Gauss -test # rep:2 # stdout:output # outputFile: % app:run.bat # par: 1000 Gauss -test # rep:2 # stdout:output # outputFile:

group2= app:fileGen.jar # par: testFolder # rep:2 # stdout:output # outputFile: testFolder % app:fileGen.jar # par: testFolder # rep:2 # stdout:output # outputFile: testFolder % app:fileGen.jar # par: testFolder # rep:2 # stdout:output # outputFile: testFolder