**4AHIT**

05.12.2013

**Distributed Pi Calculator**

**Protokoll**

**Belinic Vennesa & Scholz Dominik**

**Inhaltsangabe**

[Git Link 1](#_Toc373432769)

[Aufgabenstellung 1](#_Toc373432770)

[Designüberlegung 3](#_Toc373432771)

[Arbeitsaufteilung 4](#_Toc373432772)

[Aufwandschätzung 5](#_Toc373432773)

[Endzeitaufteilung 5](#_Toc373432774)

[Arbeitsdurchführung 6](#_Toc373432775)

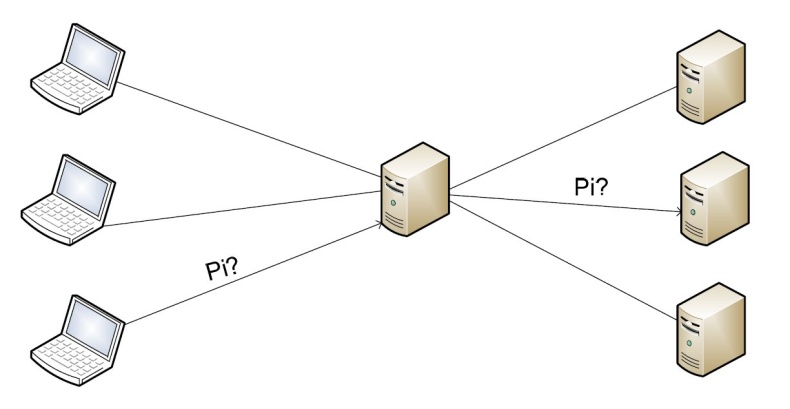
[Testbericht 7](#_Toc373432776)

[Quellenangaben 9](#_Toc373432777)

# Git Link

https://github.com/dscholz-tgm/PiCalculator  
(Dieses Repository ist erst nach dem Schließen der Abgabe sichtbar.)

# Aufgabenstellung



Als Dienst soll hier die beliebig genaue Bestimmung von pi betrachtet werden. Der Dienst stellt folgendes Interface bereit:

|  |  |
| --- | --- |
| 1  2  3  4 | *// Calculator.java*  **public** **interface** Calculator {  **public** [BigDecimal](http://www.google.com/search?hl=en&q=allinurl%3Abigdecimal+java.sun.com&btnI=I%27m%20Feeling%20Lucky) pi (**int** anzahl\_nachkommastellen);  } |

Ihre Aufgabe ist es nun, zunächst mittels Java-RMI die direkte Kommunikation zwischen Klient und Dienst zu ermöglichen und in einem zweiten Schritt den Balancierer zu implementieren und zwischen Klient(en) und Dienst(e) zu schalten. Gehen Sie dazu folgendermassen vor:

1. Ändern Sie Calculator und CalculatorImpl so, dass sie über Java-RMI von aussen zugreifbar sind. Entwicklen Sie ein Serverprogramm, das eine CalculatorImpl-Instanz erzeugt und beim RMI-Namensdienst registriert. Entwicklen Sie ein Klientenprogramm, das eine Referenz auf das Calculator-Objekt beim Namensdienst erfragt und damit pi bestimmt. Testen Sie die neu entwickelten Komponenten.
2. Implementieren Sie nun den Balancierer, indem Sie eine Klasse CalculatorBalancer von Calculator ableiten und die Methode pi() entsprechend implementieren. Dadurch verhält sich der Balancierer aus Sicht der Klienten genauso wie der Server, d.h. das Klientenprogramm muss nicht verändert werden. Entwickeln Sie ein Balanciererprogramm, das eine CalculatorBalancer-Instanz erzeugt und unter dem vom Klienten erwarteten Namen beim Namensdienst registriert. Hier ein paar Details und Hinweise:
   * Da mehrere Serverprogramme gleichzeitig gestartet werden, sollten Sie das Serverprogramm so erweitern, dass man beim Start auf der Kommandozeile den Namen angeben kann, unter dem das CalculatorImpl-Objekt beim Namensdienst registriert wird. dieses nun seine exportierte Instanz an den Balancierer übergibt, ohne es in die Registry zu schreiben. Verwenden Sie dabei ein eigenes Interface des Balancers, welches in die Registry gebinded wird, um den Servern das Anmelden zu ermöglichen.
   * Das Balancierer-Programm sollte nun den Namensdienst in festgelegten Abständen abfragen um herauszufinden, ob neue Server Implementierungen zur Verfügung stehen.
   * Java-RMI verwendet intern mehrere Threads, um gleichzeitig eintreffende Methodenaufrufe parallel abarbeiten zu können. Das ist einerseits von Vorteil, da der Balancierer dadurch mehrere eintreffende Aufrufe parallel bearbeiten kann, andererseits müssen dadurch im Balancierer änderbare Objekte durch Verwendung von synchronized vor dem gleichzeitigen Zugriff in mehreren Threads geschützt werden.
   * Beachten Sie, dass nach dem Starten eines Servers eine gewisse Zeit vergeht, bis der Server das CalculatorImpl-Objekt erzeugt und beim Namensdienst registriert hat sich beim Balancer meldet. D.h. Sie müssen im Balancierer zwischen Start eines Servers und Abfragen des Namensdienstes einige Sekunden warten.

Testen Sie das entwickelte System, indem Sie den Balancierer mit verschiedenen Serverpoolgrössen starten und mehrere Klienten gleichzeitig Anfragen stellen lassen. Wählen Sie die Anzahl der Iterationen bei der Berechung von pi entsprechend gross, sodass eine Anfrage lang genug dauert um feststellen zu können, dass der Balancierer tatsächlich mehrere Anfragen parallel bearbeitet.

Gruppenarbeit

Die Arbeit ist als 2er-Gruppe zu lösen und über das Netzwerk zu testen! Nur localhost bzw. lokale Testzyklen sind unzulässig und werden mit 6 Minuspunkten benotet!

Benotungskriterien

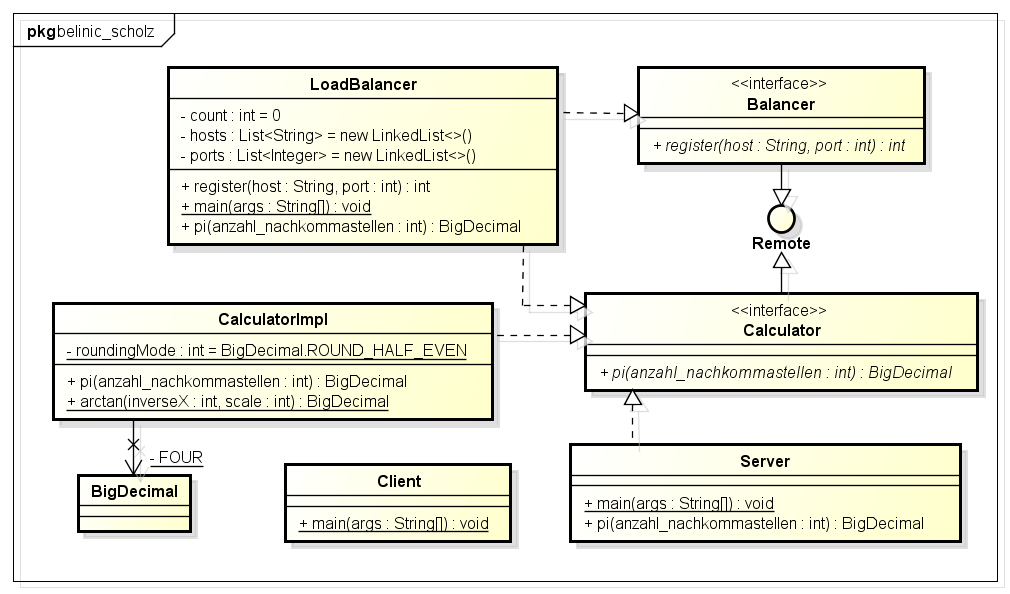
* + 12 Punkte: Java RMI Implementierung (siehe Punkt 1)
  + 12 Punkte: Implementierung des Balancers (siehe Punkt 2)
  + davon 6 Punkte: Balancer
  + davon 2 Punkte: Parameter - Name des Objekts
  + davon 2 Punkte: Listing der Server (dyn. Hinzufügen und Entfernen)
  + davon 2 Punkte: Testprotokoll mit sinnvollen Werten für Serverpoolgröße und Iterationen

Quellen

An Overview of RMI Applications, Oracle Online Resource, <http://docs.oracle.com/javase/tutorial/rmi/overview.html> (last viewed 21.11.2013)

# Designüberlegung

Da in der Angabe ein Link auf das Java Tutorial angegeben ist, ist geplant sich das Beispiel aus dem Tutorial anzuschauen, und falls möglich Sourcecode von dort zu übernehmen.

Als erste erstellten wir gemeinsam ein UML-Diagramm.

Wie bereits oben erwähnt ist es geplant Sourcecode aus dem Java Tutorial zu übernehmen. In diesem Fall würde wir die Methode zum Berechnen von Pi übernehmen und die Implementierung vom Client und vom Server.

Uns war klar das der Loadbalancer Server als auch Client zugleich sein muss, da er für den Client ein Server ist und für den Server ein Client. Daher war es irgendwie notwendig, dass sich die Server irgendwie beim Loadbalancer anmelden müssen.

# Arbeitsaufteilung

Die Arbeit wird so aufgeteilt, dass jedes Gruppenmitglied, soweit wie möglich, von einander unabhängig arbeiten kann. Deshalb werden den jeweiligen Gruppenmitglieder ganze Klassen als Arbeitspakete zugeteilt. Jedes Gruppenmitglied testes die von ihm/ihr geschriebenen Klassen so weit es nötig ist (mit JUnit4).

Vennesa

* Client
* Calculator
* CalculatorImpl
* Policy
* MyCLI
* Start

Dominik

* Server
* CalculatorBalancer

|  |  |  |
| --- | --- | --- |
|  | *geschätzte Zeit* | *tatsächliche Zeit* |
| **Belinic Vennesa** | 2h 20 min | 2 h 20 min |
| **Scholz Dominik** | 2 h 35 min | 3 h 35 min |

# Aufwandschätzung

|  |  |  |  |
| --- | --- | --- | --- |
| Klasse | Testklasse | Aufwand für Klassen | Aufwand für Testkassen |
| Calculator | nicht notig weil Interface | 5 min | - |
| Balancer | nicht notig weil Interface | 5 min | - |
| LoadBalancer | TestBalancer | 1 h 30 min | 1 h |
| CalculatorImpl | TestCalculator | 30 min | 1 h |
| Client | TestClient | 1 h | 1 h |
| Policy | nicht nötig | 15 min | - |
| Server | TestServer | 1 h | 1 h |
| Start | nicht nötig | 15 min | - |
| MyCLI | TestMyCLI | 15 min | 45 min |
| Summe | | 5 h | 4 h 45 min |
| Gesamtsumme | | **9 h 35 min** | |

# Endzeitaufteilung

|  |  |  |  |
| --- | --- | --- | --- |
| Klasse | Testklasse | Aufwand für Klassen | Aufwand für Testkassen |
| Calculator | nicht notig weil Interface | 5 min | - |
| Balancer | nicht notig weil Interface | 5 min | - |
| LoadBalancer | TestBalancer | 2 h | - |
| CalculatorImpl | TestCalculator | 20 min | - |
| Client | TestClient | 40 min | - |
| Policy | nicht nötig | 15 min | - |
| Server | TestServer | 1 h 30 min | - |
| Start | nicht nötig | 15 min | - |
| MyCLI | TestMyCLI | 45 min | - |
| Summe | | 5 h 55 min | - |
| Gesamtsumme | | **5 h 55 min** | |

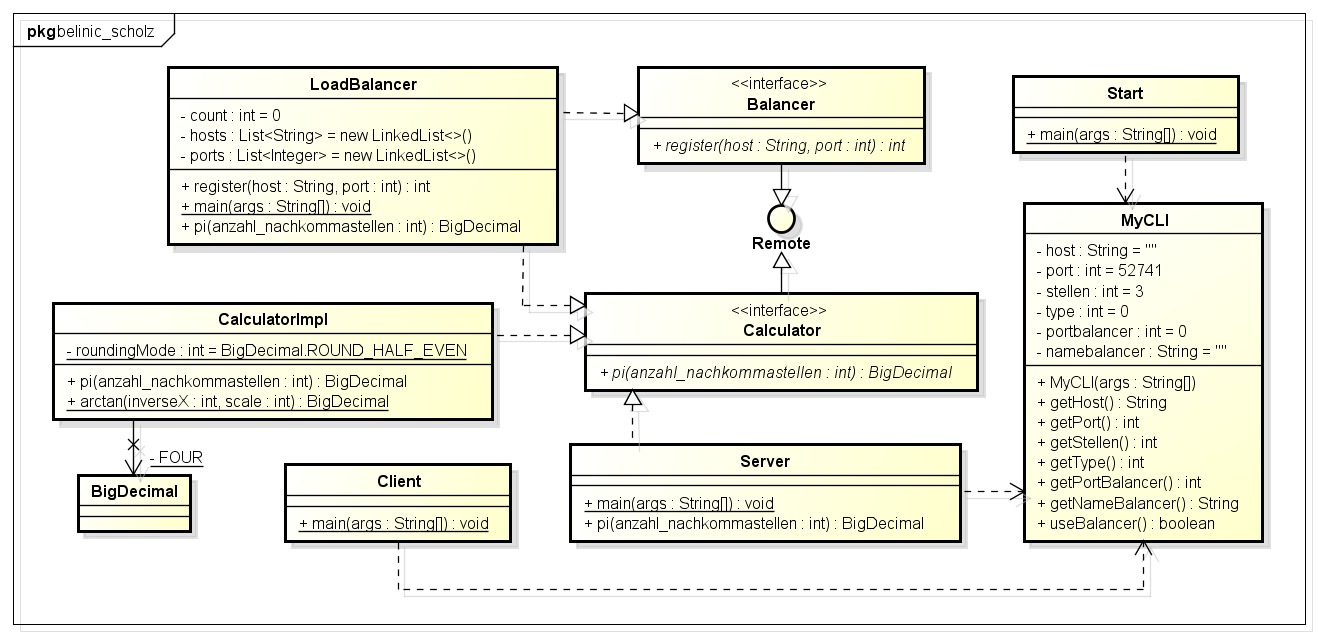
# Arbeitsdurchführung

Bei der Arbeitsdurchführung stießen wir vorerst auf kleine Problem, es war nicht ganz klar was als codeBase in der Policy anzugeben war, doch nach Rücksprache mit dem Lehrer ergab sich, dass es mit unseren eigenen (lokalen) Policy funktionieren soll.

Dann kamen wir darauf, dass wir eine Verwaltung der Optionen und Argumente benötigten, und dass es besser wäre eine Startklasse zu implementieren, die je nachdem was der User eingibt den Balancer den Server oder den Client startet.

Nachdem Server und Client implementiert waren, versuchten wir den Sourcecode auszuführen, doch dies funktionierte nicht. Wir erhielten ständig eine Java ExportException, bei der Suche nach dem Bug, fanden wir heraus, dass lediglich ein throws bei der Methode pi(int), vom Interface Calculator, fehlte.

Als dieses Problem behoben war, kam das nächte Problem, irgendetwas stimmt mit der Policy nicht. Da wir nichts in unseren lokalen Policy ändern wollten, haben wir einen eigene Policy angefertigt die beim Ausführen mitangegeben werden muss (-Djava.security.policy=file) [1]

**Neues UML-Diagramm**

# Testbericht

**Alles auf dem gleichen Rechner**

zuerst starte ich das rmiregistry:



dann starte ich den Balancer:



danach starte ich einige Server:



danach starte ich mehre Clients:



**Auf dem gleichen Rechner, aber Server und Balancer auf einer VM:**

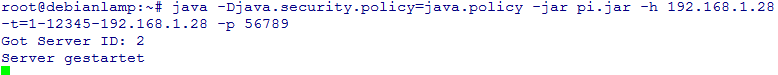
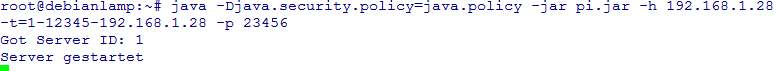
zuerst starte ich das rmiregistry (Debian):



danach starte ich den Balancer (Debian):



danach starte ich mehrer Server (Debian)



danach starte ich mehre Clients auf meinem "echten" OS (WIndows):



# Quellenangaben

1. Java, "Running the Example Programs", aktualisiert: 2013,online verfügbar: http://docs.oracle.com/javase/tutorial/rmi/running.html, entnommen am: 04.12.2013, zuletzt besucht am: 05.12.2013