Praktische Übung 2 zur Vorlesung "Verteilte Systeme" ETH Zürich, SS 2000

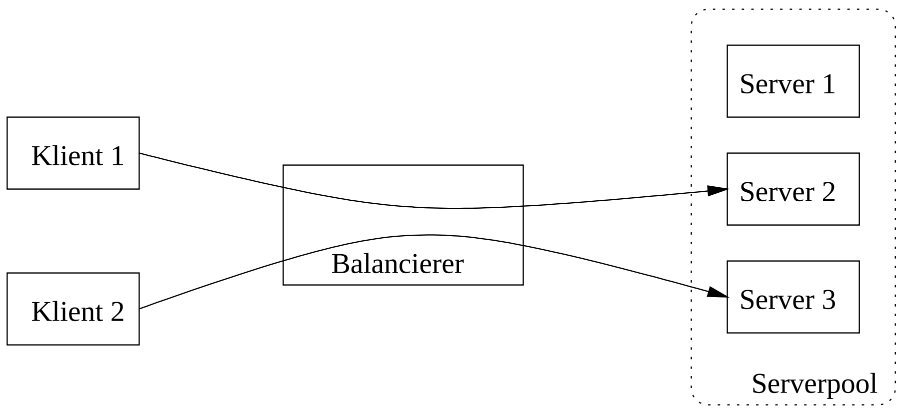
Prof. Dr. F. Mattern
Assistenten: K. Römer, S. Domintcheva

Ausgabedatum: 2. Mai 2000
Abgabedatum 24. Mai 2000

Nachdem Sie in Aufgabe 1 selbst ein einfaches RPC-System implementiert haben, geht es in dieser Aufgabe darum, unter Verwendung von RMI (*Remote Method Invocation*, dem von Java bereitgestellten RPC-Mechanismus) ein System zur *automatischen Lastverteilung* zu entwickeln.

Einführung

Man kann die Leistungsfähigkeit (und auch die Ausfallsicherheit) eines verteilten Systems erhöhen, indem man einen Dienst gleichzeitig auf mehreren Rechnern (Servern) anbietet und Anfragen von Klienten jeweils an die Dienstinkarnation mit der geringsten Auslastung delegiert. Ein solches *Lastverteilungssystem* besteht im einfachsten Fall wie im Bild dargestellt aus einer Menge von Servern (dem sogenannten *Serverpool*), die alle ein und den selben Dienst anbieten, sowie einem *Balancierer*, der von Klienten eintreffende Anfragen an den Server mit der geringsten Auslastung weiterleitet.



Um zu entscheiden, welcher Server derjenige mit der geringsten Auslastung ist, wird ein sogenannter *Lastmonitor* benötigt, der die Auslastung der Server ständig überwacht. Wenn die Dienste einen Zustand besitzen, d.h. wenn das Ergebnis einer Anfrage an einen Dienst von den vorhergehenden Anfragen an diesen Dienst abhängt, müssen die Zustände der verschiedenen Dienstinkarnationen ständig abgeglichen (*synchronisiert*) werden. Andernfalls würde das Ergebnis einer Anfrage davon abhängen, welche Dienstinkarnation der Balancierer auswählt. Zusätzliche Probleme treten auf, wenn die Dienste *nichtdeterministisch* sind, d.h. Anfragen an einen Dienst mit exakt gleichen Parametern und Zuständen verschiedene Ergebnisse liefern können.

In dieser Aufgabe wollen wir das einfachste Szenario betrachten: deterministische, zustandslose Dienste, d.h. es ist keine Synchronisation der Dienstinkarnationen notwendig. Zudem wird ein Lastverteilungsverfahren (*Round-Robin*) verwendet, das keinen Lastmonitor erfordert. Bei Round-Robin werden die von Klienten eintreffenden Dienstanfragen einfach reihum an die zur Verfügung stehenden Server verteilt.

Aufgabenstellung

Als Dienst soll hier die beliebig genaue Bestimmung von π betrachtet werden[1]. Der Dienst stellt folgendes Interface bereit:

```
// Calculator.java

public interface Calculator {
    public double pi (int anzahl_iterationen);
}
```

Wir stellen Ihnen dieses Interface und die dazugehörige lokale Implementierung (`CalculatorImpl`) bereit. 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 π bestimmt. Testen Sie die neu entwickelten Komponenten.

2. Implementieren Sie den Balancierer, indem Sie eine Klasse `CalculatorBalancierer` 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 `CalculatorBalancierer`-Instanz erzeugt und unter dem vom Klienten erwarteten Namen beim Namensdienst registriert. Hier einige Hinweise und Details:

    - 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.
    - Das Balancierer-Programm sollte über einen Kommandozeilenparameter verfügen, mit dem man die Anzahl der Server im Serverpool steuern kann. Das Balanciererprogramm sollte beim Start entsprechend viele Serverprogramme starten.

[1] Mittels der konvergenten Reihe $\pi = 4 \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1}$. Es gibt zugegebenermassen wesentlich effizientere Wege zur Bestimmung von π, allerdings geht es hier einfach darum, einen halbwegs sinnvollen Dienst anzubieten, der relativ lange Zeit zur Ausführung benötigt.

- 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[2].
- Beachten Sie, dass nach dem Starten eines Servers eine gewisse Zeit vergeht, bis der Server das `CalculatorImpl`-Objekt erzeugt und beim Namensdienst registriert hat. D.h. Sie müssen im Balancierer zwischen Start eines Servers und Abfragen des Namensdienstes einige Sekunden warten.

3. 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 π entsprechend gross, so dass eine Anfrage lang genug dauert um feststellen zu können, dass der Balancierer tatsächlich mehrere Anfragen parallel bearbeitet.

Abschliessend noch einige nützliche Java-Methoden, die für die Lösung der Aufgabe hilfreich sind[3]:

- `Integer.toString(int i)` zur Umwandlung eines Integers in einen String.
- `Integer.parseInt(String s)` zur Umwandlung eines Strings in einen Integer.
- `Runtime.getRuntime().exec(String kommando)` zum Ausführen eines Programms, wobei `kommando` der Eingabe in eine Unix-Shell entspricht, z.B. "`java Server`". Leider werden die Ausgaben (beispielsweise mittels `System.out.println()` in einem Java-Programm) eines derart gestarteten Programms nicht sichtbar. Sie können sich jedoch helfen, indem Sie wie folgt einen `PrintStream` erzeugen und anstelle von `System.out` benutzen:

```
PrintStream out = new PrintStream (new FileOutputStream ("/dev/tty"));
out.println ("hello world");
```

- `Thread.sleep(int ms)` um `ms` Millisekunden zu warten.

[2] Objekte, die nur gelesen aber nicht geändert werden, müssen natürlich nicht geschützt werden.
[3] Details zu diesen Klassen finden Sie in der Java API-Dokumentation unter `http://java.sun.com/products/jdk/1.2/docs/api/overview-summary.html`.