JUST DO IT
HÖHERE ABTEILUNG FÜR
INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Laborprotokoll

Remote Method Invocation

Systemtechnik Labor
4AHIT 2017/18, GruppeA

Marc Rousavy

Version 0.2
Note:
Betreuer: M. Borko
Begonnen am 9. September 2017
Beendet am 23. Januar 2018

Inhaltsverzeichnis

1 Einführung

Verteilte Objekte haben bestimmte Grunderfordernisse, die mittels implementierten Middlewares leicht verwendet werden können. Das Verständnis hinter diesen Mechanismen ist aber notwendig, um funktionale Anforderungen entsprechend sicher und stabil implementieren zu können.

1.1 Ziele

Diese Übung gibt eine einfache Einführung in die Verwendung von verteilten Objekten mittels Java RMI. Es wird speziell Augenmerk auf die Referenzverwaltung sowie Serialisierung von Objekten gelegt. Es soll dabei eine einfache verteilte Applikation in Java implementiert werden.

1.2 Voraussetzungen

- Grundlagen Java
- Grundlagen zu verteilten Systemen und Netzwerkverbindungen
- Grundlegendes Verständnis von nebenläufigen Prozessen

1.3 Aufgabenstellung

Folgen Sie dem offiziellen Java-RMI Tutorial, um eine einfache Implementierung des PI-Calculators zu realisieren. Beachten Sie dabei die notwendigen Schritte der Sicherheitseinstellungen (Security-Manager) sowie die Verwendung des RemoteInterfaces und der RemoteException.

Als Implementierungsgrundlage dient folgender Classroom-Link.

Implementieren Sie einen weiteren Task nach dem Command-Pattern [2] mittels RMI und übertragen Sie die Berechnung an den Server. Erweitern Sie die Implementierung des Tutorials ohne große Anpassungen. Erstellen Sie zum Beispiel einen Task zur berechnung der Fibonacci-Folge [3] und führen Sie diesen nebst der Pi-Berechnung aus.

Die Erweiterung dieser Aufgabe wäre ein Loadbalancer-Interface auf der Server-Seite, die Anfragen an eine Compute-Instanz an weitere Server weiterleitet. Der Client soll dabei gar nicht geändert werden. Es soll möglich sein, dass mehrere Server sich beim Loadbalancer registrieren können und für Berechnungen (computeTask) zur Verfügung stehen. Der Loadbalancer hat dabei nur eine verwaltende Tätigkeit zu übernehmen und erscheint für den Client weiterhin als Implementierung des Compute-Interfaces. Es bleibt Ihnen überlassen, wie die Verwaltung der Server-Stubs beim Loadbalancer umgesetzt wird. Es ist eine einfache Round-Robin-Verteilung zu implementieren.

Die Implementierung soll grundsätzlich auch über die Systemgrenzen funktionstüchtig sein (Achtung wegen RMI-Registry und Verwendung der RMI-Stubs).

1.4 Bewertung

Die Abgabe erfolgt durch das Repository. Bitte nicht auf regelmäßige Commits mit sprechenden Commitmessages vergessen. Zur Bewertung soll das Protokoll folgendermaßen ausgedruckt werden: "Hochformat; 2 Seiten pro Blatt und über die lange Seite drehen" (damit kommen vier Seiten auf ein Blatt A4).

- Gruppengrösse: 1 Person
- Anforderungen "Grundkompetenz"
  - Dokumentation und Beschreibung anhand der Protokollrichtlinien
  - Java RMI-Tutorial um "sauberes Schließen" erweitern
  - Implementierung eines neuen Tasks (z.B. Fibonacci)
  - Implementierung eines Loadbalancer-Interfaces (register/unregister)
- Anforderungen "Erweiterte Kompetenz"
  - Client-Loadbalancer-Server-Verbindungen über mehrere Rechner hinweg lauffähig (z.B. mittels Portweiterleitung)
  - Überlegungen zum Design und mögliche Implementierung weiterer Loadbalancing-Methoden (Weighted Distribution oder Least Connections)

1.5 Quellen

1. "The Java Tutorials - Trail RMI"
2. "Command Pattern", Vince Huston
3. "Fibonacci Number Program", wikibooks

2 Ergebnisse

Der Grundaufbau besteht aus einem Client Java Programm, welches über das Remote Method Invocation ("RMI") Protokoll einen Server kontaktiert, welcher dann wiederum eine beliebige Aufgabe für diesen Client ausführt.

Diese Aufgabe ("Task") soll generisch sein, sodass verschiedene Implementationen von beliebigen Aufgaben ausgeführt werden können, wie beispielsweise die Berechnung von Pi oder der Fibonacci Nummer.

Um die Aufgabe zu erweitern, wird ein Proxy zwischen die Clients und Server geschalten, welcher dann einzelne Anfragen der Clients mittels Round Robin Konzept gleichgerecht auf die einzelnen Server verteilt.

2.1 Projekt

Als Entwicklungsumgebung wird IntelliJ IDEA von JetBrains [1] verwendet. Es wird ein neues Projekt erstellt, mit folgenden packages/submodules:

- **Client**: Client-side RMI implementierung
- **Server**: Server-side RMI implementierung, inkl. Load Balancer
- **Modules**: Shared-code, u.a. die generische **Task<T>** klasse, Pi und Fibonacci implementierungen, etc.

2.2 Module

Bevor der Client oder Server implementiert werden kann, muss eine basis gebaut werden. Es werden Funktionen implementiert, welche Client **und** Server verwenden. Unter anderem gehört dazu:

2.2.1 Compute

Es muss ein Interface definiert werden, welches als Stub [1] dient, also die Schnittstelle zur Implementierung für den **Task** auf einer anderen Codebase.

Die Implementierung ist sehr simpel, es besteht aus einem Interface mit der **run()** Funktion:

```
1 public interface Compute extends Remote, Serializable {
      <T> T run(Task<T> t) throws RemoteException;
3 }
```

Listing 1: Module Implementation - Compute interface

2.2.2 Task

Es muss ein Generisches Interface definiert werden, welches auch nur eine **run()** Funktion aufweist:

```
1 public interface Task<T> {
      T run();
3 }
```

Listing 2: Module Implementation - Task interface

Dieses Interface ist dazu da, um jeden beliebigen Auftrag auszuführen, ohne die Implementierung (bzw. **run()**-Methode) zu kennen.

2.2.3 Fibonacci

Nun kann eine Tatsächliche Aufgabe implementiert werden, als erstes Beispiel wird Fibonacci verwendet.

Die Klasse Fibonacci soll aus **Task<BigInteger>** erben da es ein Task mit dem Rückgabewert BigInteger ist, sowie aus **Serializable**, damit aus dem Memory void* ein serialisiertes byte array gebaut werden kann.

Das serialisierte byte array kann somit über die Netzwerkschnittstelle gesendet werden.

Für die Serilization wird außerdem eine statische, final Konstante aus einem 64bit Integer erstellt, welche quasi einen Identifier darstellt welcher der Serilisation erkennbar gibt, dass es sich um die selbe Implementierung einer Klasse handelt.

Die **run()** Methode schaut folgendermaßen aus:

```
1 @Override
  public BigInteger run() {
3     BigInteger previous = BigInteger.ONE;
      BigInteger recent = BigInteger.ONE;
5     for (int i = 0; i < _digits; i++) {
          BigInteger temp = previous.add(recent);
7         previous = recent;
```

```
            recent = temp;
        }
        return recent;
    }
```

Listing 3: Module Implementation - Fibonacci run

2.2.4 Pi

Die Implementation für die Berechnung von Pi ist etwas komplizierter, hierbei wird ein Tutorial von Oracle verwendet (leicht abgeändert):

```
@Override
public BigDecimal run() {
    int scale = _digits + 5;
    BigDecimal arctan1_5 = arctan(5, scale);
    BigDecimal arctan1_239 = arctan(239, scale);
    BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
                    arctan1_239).multiply(FOUR);
    return pi.setScale(_digits,
                       BigDecimal.ROUND_HALF_UP);
}
```

Listing 4: Module Implementation - Pi run

Wobei die arctan folgende Implementation hat:

```
public static BigDecimal arctan(int inverseX,
                                 int scale) {
    BigDecimal result, numer, term;
    BigDecimal invX = BigDecimal.valueOf(inverseX);
    BigDecimal invX2 =
    BigDecimal.valueOf(inverseX * inverseX);
    numer = BigDecimal.ONE.divide(invX,
                                  scale, roundingMode);
    result = numer;
    int i = 1;
    do {
        numer = numer.divide(invX2, scale, roundingMode);
        int denom = 2 * i + 1;
        term = numer.divide(BigDecimal.valueOf(denom), scale, roundingMode);
        if ((i % 2) != 0) {
            result = result.subtract(term);
        } else {
            result = result.add(term);
        }
        i++;
    } while (term.compareTo(BigDecimal.ZERO) != 0);
    return result;
}
```

Listing 5: Module Implementation - Pi arctan

2.3 Client

Es wird ein simpler Client implementiert, welcher den Server (bzw. Proxy) mittels RMI kontaktieren soll.

Als implementierung muss in der System Registry nach dem Stub [2] gesucht werden, welcher die Schnittstelle zu dem Server (bzw. Proxy) darstellt.

```
1 public Client(String host, String stubName) throws RemoteException, NotBoundException {
      _registry = LocateRegistry.getRegistry(host);
3     _compute = (Compute) _registry.lookup(stubName);
  }
```

Listing 6: Client Implementation - Registry lookup

Der Client kann mit dem gefundenen Stub nun jeden beliebigen Task ausführen:

```
  public <T> T run(Task<T> task) throws RemoteException {
2     return _compute.run(task);
  }
```

Listing 7: Client Implementation - Task run

2.3.1 Beispiel

Als Beispiel kann dieser Client die Fibonacci Number berechnen lassen:

```
1 Fibonacci fibonacci = new Fibonacci(5);   // Fibonacci Number bis 5
  BigInteger number = client.run(fibonacci);
```

Listing 8: Client Implementation - Fibonacci Beispiel

2.4 Server

Es wird ein basis-Interface definiert, welches den Zugriff für den Proxy später vereinfachen wird. Das Interface wird **Processor** genannt, und erweitert die **Compute** Klasse. Der Interface definiert nur eine Funktion: **busy()**, welche einen boolean wert zurückliefer.

Der Server ist die Implementierung der **Processor** Klasse, welche eine **run()** Methode besitzt um einen beliebigen Task<T> auszuführen, und das Ergebnis zurück zu liefern.

Für die **busy()** Methode wird eine Variable namens **busy** gespeichert, welche besagt, ob der Server gerade beschäftigt ist oder nicht.

```
@Override
public <T> T run(Task<T> task) throws RemoteException {
    try {
        _busy = true;
        T result = task.run();
        JLogger.Instance.Log(Logger.Severity.Info, "Executed Task \"" + task.toString() + "\", at " +
            new Date().toString());
        return result;
    } finally {
        _busy = false;
    }
}
```

Listing 9: Server Implementation - run Methode

2.5 Proxy

Nun kann ein Proxy zwischen Client und Server geschaltet werden.

2.5.1 Idee

Der Proxy, welcher für den Client als ein einziger Server erscheint, dient dazu, die (mehreren-) Anfragen von Clients gleichmäßig auf registrierte Server zu verteilen.

Umgesetzt wird das ganze mittels dem Round Robin Konzept, welches den Servern jeweils eine Aufgabe gibt. Sobald jeder Server einmal gearbeitet hat, wird der Zyklus zurückgesetzt und es wird wieder bei dem ersten Server angefangen.

2.5.2 Implementierung

Es wird ein Interface definiert, welches den Round Robin Load Balancer Proxy beschreibt.

```
1 public interface LoadBalancer extends Compute {
      void add(Processor processor) throws RemoteException;
3     boolean remove(Processor processor) throws RemoteException;
  }
```

Listing 10: Proxy Implementation - LoadBalancer Interface

Das Interface erweitert Compute, da es ja Aufgaben (Tasks) berechnet (run).

Außerdem ist deutlich erkennbar, dass das Interface exakt wie ein normaler Server ausschaut, da Clients nicht wissen müssen, dass Sie mit einem Proxy kommunizieren.

Als nächstes wird das Interface in einer Klasse **"Proxy"** implementiert. Es werden folgende Objekt-Member erstellt:

- logger: Ein beliebiger Logger, hierbei wird er zu dem System.out Stream weitergeleitet.
- processors: Eine List<Processor> welche alle registrierten Prozessoren (Server) speichert.
- iter: Ein Iterator<Processor> zu der Liste aus Prozessoren, welcher den Fortschritt in einem Round Robin Zyklus speichert.

Die Methoden **add()** und **remove()** sind sehr trivial, da sie hierbei nur die processors liste bearbeiten, und den iterator zurücksetzen;

```
  @Override
2 public void add(Processor processor) throws RemoteException {
      _processors.add(processor);
4     _iter = _processors.iterator();
      _logger.Log(Logger.Severity.Info,
6             "Registered new Server: \"" + processor + "\"");
  }
```

Listing 11: Proxy Implementation - Proxy::add

```
@Override
public boolean remove(Processor processor) throws RemoteException {
    boolean success = _processors.remove(processor);
    _iter = _processors.iterator();
    if(success)
        _logger.Log(Logger.Severity.Info,
                "Removed Server: \"" + processor + "\"");
    return success;
}
```

Listing 12: Proxy Implementation - Proxy::remove

Und nun kann die **run()** Methode des Proxy implementiert werden, welche auch nur den Task einem Server weitergibt;

```
@Override
public <T> T run(Task<T> t) throws RemoteException {
    do {
        if (_iter.hasNext())
            _iter = _processors.iterator();

        Processor p = _iter.next();
        if (!p.busy()) {
            return p.run(t);
        }
    } while(true);
}
```

Listing 13: Proxy Implementation - Proxy::run

Der Iterator wird überprüft ob er am Ende angelangt ist, falls dies der Fall ist, wird er zurückgesetzt. Danach wird ein Prozessor (Server) aus der Liste geholt, und der Iterator weitergeschoben. Falls dieser Prozessor nicht gerade beschäftigt ist, kann der Task an den Prozessor weitergereicht werden, und auf ein Ergebnis gewartet werden.

Der Proxy kann auch so konfiguriert werden, dass er den Prozessoren unabhängig davon ob sie beschäftigt sind oder nicht, die Tasks weitergibt, wodurch aber viel beschäftigte Server eine Lange Arbeitskette (Task Queue/Stack) haben könnten.

2.6 Glossar

1. ”IntelliJ IDEA - JetBrains”
2. ”Stub:” eine Schnittstelle zu Funktionen oder Prozeduren, welche auf einem anderen System implementiert sind.

Tabellenverzeichnis

Listings

Abbildungsverzeichnis