UNIVERSITÄT SIEGEN

Fakultät IV
Betriebssysteme und verteilte Systeme
Prof. Dr. rer. nat. Roland Wismüller

Aufgabenblatt 4
Musterlösung

Vorlesung Verteilte Systeme
Sommersemester 2020

Aufgabe 1: Programmierung: Java-RMI - Folgebeispiel

Die Lösung dieser Aufgabe finden Sie im Archiv `l04Files.zip`[1] auf der Vorlesungswebseite.

Aufgabe 2: Programmierung: Generische Proxy-Objekte

Die Lösung dieser Aufgabe finden Sie im Archiv `l04Files.zip`[2] auf der Vorlesungswebseite.

Aufgabe 3: Generischer Dispatcher mit Java Reflection

a) Die Anfrage-Nachricht muss in jedem Fall folgende Informationen enthalten:

(i) einen Identifikator für das Objekt, auf dem die Methode aufgerufen werden soll,

(ii) eine Spezifikation der aufzurufenden Methode,

(iii) die Argumente für den Methodenaufruf.

Die Objekt-ID kann im einfachsten Fall eine fortlaufende Zahl sein. Das Server-Skeleton muß dann z.B. ein Array pflegen, das Referenzen auf alle Remote-Objekte des Servers enthält.

Die aufzurufende Methode könnte über ihren Namen angegeben werden. Wegen der Möglichkeit des Überladens reicht das aber evtl. nicht aus, sondern es müssen zusätzlich auch noch die Parametertypen mit spezifiziert werden. Die Java-Klasse `Class` beinhaltet eine Methode `getMethod()`, mit der zu einem gegebenen Namen und den Parametertypen das entsprechende `Method`-Objekt gefunden werden kann.[3] Da `Method` selbst nicht serialisierbar ist, kann die Nachricht das `Method`-Objekt nicht direkt beinhalten.

Die Argumente für den Methodenaufruf lassen sich in Java am einfachsten als serialisierte Objekte übertragen. Die Argumente müssen bei RMI immer entweder serialisierbare oder Remote-Objekte sein. Im letzteren Fall wird beim Aufruf das zugehörige Stub-Objekt serialisiert übergeben.

Der Java-Datentyp für eine Anfragenachricht könnte also wie folgt aussehen:

```
public class RequestMessage implements Serializable
{
   int         objectID;
   String      methodName;
   Class<?>[] argumentTypes;
   Object[]    arguments;
   ...
}
```

[1] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/l04Files.zip`
[2] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/l04Files.zip`
[3] Dabei müssen die Parametertypen aber exakt mit den spezifizierten Typen übereinstimmen. Prinzipiell könnte man die Methode auch automatisch aus den Typen der übergebenen Argumente ermitteln. Diese können aber Unterklassen der Parametertypen sein, so daß die Suche nach der passenden Methode nicht trivial ist. Die Java-Klasse `Class` bietet dafür auch keine Methode an.

b) Mit dem oben genannten Typ für die Anfragenachricht könnte der Code für einen generischen Dispatcher etwa so aussehen:

```
public void dispatch(RequestMessage request)
{
   try {
      // Referenz auf das Zielobjekt aus objectID bestimmen
      Object target = objectTable[request.objectID];
      // Über Class-Objekt das passende Method-Objekt bestimmen
      Method method = target.getClass().getMethod(request.methodName,
                                                  request.argumentTypes);
      // Methode aufrufen
      Object result = method.invoke(target, request.arguments);
      ... result in Antwortnachricht verpacken und zurücksenden
   }
   catch (InvocationTargetException e) {
      // Aufgerufene Methode warf eine Exception
      ... e.getCause() in Antwortnachricht verpacken und zurücksenden
   }
   catch (Exception e) {
      // Exception beim Aufruf der Methode (z.B. IllegalArgumentException)
      ... e in Antwortnachricht verpacken und zurücksenden
   }
}
```

Aufgabe 4: Parameterübergabe

Wird *call-by-reference* verwendet, wird `incr` ein Zeiger auf `i` übergeben. Damit wird `i` wird zweimal inkrementiert, das Endergebnis ist also in diesem Fall 2.

Bei *call-by-value* wird der Wert direkt übergeben und keine Referenz auf `i`. Daher bleibt `i` beim Aufrufer unverändert, also 0.

Call-by-copy/result bedeutet: Kopieren der Parameterwerte beim Aufruf der Prozedur, Zurückkopieren und Überschreiben der Aufrufparameter bei Beendigung der Prozedur. In diesem Fall wird `i` als zweimal als Wert übergeben (also jeweils 0) und jeweils inkrementiert, so dass beide Werte nun 1 sind. Wenn sie zurückkopiert werden, überschreibt die zweite Kopie die erste und der endgültige Wert ist nun ebenfalls 1.

Aufgabe 5: Transparenz von Java RMI

Die Lösungen der Programmieraufgabe finden Sie im Archiv `l04Files.zip`[4] auf der Vorlesungswebseite. Das Problem ist die Parameterübergabe bei Java RMI, die für serialisierbare Objekte (und `ArrayList` ist serialisierbar) eine *call by value* Semantik hat. D.h., der Server sortiert zwar korrekt die Liste, das Ergebnis hat aber keinen Einfluß auf den Client.

Um das Problem zu reparieren, könnte man auf die Idee kommen, aus der Liste ein Remote-Objekt zu machen, das beim Client liegt, so daß der Server diese über RMI verändern kann. Da beim Sortieren einer `ArrayList` jedoch extrem viele Zugriffe entstehen, wäre dieses Vorgehen extrem ineffizient. Effizienter ist es daher, eine Parameterübergabe mit *call by value and result* Semantik zu realisieren, indem die `sort()`-Methode der Remote-Schnittstelle `SortServer` die sortierte Liste als Rückgabewert liefert und `sort()`-Methode der Wrapper-Klasse diesen Rückgabewert wieder in die ursprüngliche Liste kopiert.

[4] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/l04Files.zip`