Universität Dortmund
Lehrstuhl Verteilte Systeme

Dr. Michael Zapf

Michael.Zapf@cs.uni-dortmund.de
Michael.Zapf@uni-kassel.de

vkiv@ls4.cs.uni-dortmund.de

Übung zur Vorlesung „Verteilt-kooperative Informationsverarbeitung“ - SS 2007

Lösungen zu Blatt 2

Aufgabe 2.1 – 10 Punkte

Ein Virtual Home Environment [1] ist eine Unterstützung des Anwenders im UMTS-Netz, die ihm den Eindruck vermittelt, dass er auch in fremden Netzen alle seine persönlichen Dienste wiederfindet (er sich also „wie zu Hause fühlt“). Die Herausforderung besteht dabei darin, dass der Anwender auch über größere Distanzen hinweg mit möglicherweise komplexen Funktionen seiner Umgebung versorgt werden soll. Dies würde zu erheblichem Kommunikationsaufwand führen.

Wäre dies ein mögliches Einsatzgebiet für mobile Agenten? [...]

Mögliche Einsatzszenarien: Mobile Agenten könnten in zwei verschiedene Klassen von Szenarien auftauchen:

1. Mobile Agenten reisen dem Anwender hinterher. um die Funktionalität seiner gewohnten Umgebung zur Verfügung zu stellen. Anstatt mit den stationären Diensten zu kommunizieren, spricht der Anwender mit seinen lokalen oder in der Nähe befindlichen Agenten. Die vom Agent verrichtete Leistung, die auf dem Gerät (oder in seiner Nähe) angeboten wird, muss möglichst in sich abgeschlossen sein, also nur wenig Netzverbindungen erfordern. Es ist fraglich, welcher Art diese Dienste sein können; eventuell könnte man sich vorstellen, ein Kontaktverzeichnis (Telefonbuch) übertragen zu bekommen, das dann je nach Kontext, in dem sich der Anwender befindet, vom Agenten aufbereitet wird (Beispiel: Man befindet sich auf einer Konferenz; alle unwichtigen Adressen werden ausgeblendet).

2. Mobile Agenten werden vom Anwender in die Heimumgebung geschickt, migrieren also „nach Hause“ und verrichten dort eine Arbeit. Dies erlaubt es, aufwändige Operationen (mit hohem Austauschvolumen) im Heimnetz durchzuführen, ohne die Daten über das (kostenpflichtige) Netz zu übertragen. Man würde in bekannter Manier dem Agenten einen Auftrag erteilen, mehrere Aktionen selbständig auszuführen und dann nach erfolgter Arbeit wieder zurückzukehren. Dies ist jedoch nicht das eigentliche Szenario in einem VHE.

Voraussetzungen an Endgeräte und Infrastruktur: Die Geräte müssen die Ausführung von Agenten unterstützen. Anderenfalls könnte man noch vorsehen, dass ein Agent zumindest „in die Nähe“ des Geräts gelangt. Die Infrastruktur muss also geeignete Lokationsmechanismen bieten – etwa ein Dienst, der angibt, wo eine sichere Stelle zur Ausführung vorhanden ist. Ein Mechnismus zur Weiterleitung von Agenten durch fremde Netze, ein Namensdienst oder ein anderes Verfahren zur Behandlung des Problems wechselnder Netzadressen ist ebenfalls ratsam.

Andere Mechanismen: Für ein VHE sind andere Mechanismen vermutlich näher liegend. So könnte eine CoD-Technik (analog zu den Applets) dafür sorgen, dass man lokal mit Funktionen versorgt wird, wobei Änderungen dann später direkt übertragen werden. Ein REV-Ansatz ist in den Fällen denkbar, dass man eine komplexe Operation durchführen muss, die zahlreiche Netzverbindungen erforderten, wenn sie vom Gerät des Anwenders initiiert würden. Diese Operationen können dann stattdessen entfernt ausgeführt werden. Solche Szenarien scheinen allerdings zurzeit sehr konstruiert.

(Hintergrund dieser Aufgabe ist eine Abhandlung, in der analysiert wird, welche Vorzüge der Einsatz von mobilen Agenten in VHEs haben könnte. Die Autoren kommen zum Schluss, dass man in kürzerer Zeit wesentlich mehr Dienste in Anspruch nehmen kann, wenn diese dem Anwender in Form mobiler Agenten „hinterherreisen“. Zwar werden Betrachtungen aufgestellt, wie aufwändig die Migration im Vergleich zur Kommunikation „nach Hause“ ist und wie viel man im Gegenzug durch eine „kurze“ Kommunikation spart. Es werden jedoch keine konkreten Szenarien erwähnt, und es fehlte völlig die Betrachtung, dass ein Agent per se noch keinen Dienst verrichten kann, sondern dazu auf externe Partner angewiesen ist – seien es Dienste oder andere Agenten.)

Aufgabe 2.2 – 15 Punkte

Implementieren Sie in Java einen Tupelraum (mit der Standard-Java-API) und demonstrieren Sie die Funktion mittels eines einfachen Beispiels. Achten Sie darauf, dass ihre Tupelraum-Teilnehmer auf verschiedenen Rechnern laufen können. (Beachten Sie bitte die Hinweise zu Programmieraufgaben in der Übung.)

Hinweise

Der Tupelraum soll von verschiedenen Rechnern erreichbar sein. Daher sollte man einen Serverprozess einrichten, der auf einem bekannten Port Anfragen zulässt.

Um sich an den Tupelraum anzubinden, müssen die Teilnehmer die Serverposition kennen (Rechner, Port). Dann kann jeder Klient eine Socketverbindung öffnen. Es ist zu empfehlen, dass jeder Anfrager auf Serverseite mit einem eigenen Thread bedient wird. Diese Threads greifen auf eine gemeinsame Datenstruktur zu, etwa eine Liste.

Eine praktikable Lösung ist es, den Server mittels RMI anzusprechen und die RMI-Registry zur Registrierung zu verwenden.

Bei einer Anfrage *in(Liste)* muss das Tupel mit allen Tupeln in der vorhandenen Liste der Tupel am Server verglichen werden. Wenn es nicht allzu effizient laufen muss, bekommt man alle Ergebnisse in O(#Tupel) bei einer begrenzten Tupellänge. Mit Hashtabellen kann die Abfrage noch beschleunigt werden.

Eine Lösung dieser Aufgabe wird auf allgemeinen Wunsch nachgereicht und auf unseren Webseiten angeboten.

Aufgabe 2.3 – 10 Punkte

Warum kennt Java keine starke Migration? Versetzen Sie sich in die Rolle eines Entwicklers im Java Community Process, der solche Anfragen zuhauf bekommt und argumentieren Sie für oder gegen die Einführung von Mechanismen in Java, welche die Implementierung der starken Migration begünstigen. [...]

Starke Migration stellt hohe Spezialanforderungen an die Implementierung der Java-VM, dennoch kann man sich vorstellen, dass eine solche Funktion irgendwann einmal verfügbar wird.

Technisch gesehen ist indes unklar, in welcher Form eine solche Reaktivierung stattfinden soll: Es gilt nicht nur die aktuellen Positionen aller Threads wieder herzustellen; man muss auch alle Aufrufstacks wieder aufbauen. Dazu muss sichergestellt sein, dass der komplette Objektbaum wieder zur Verfügung steht (sonst würde eine Rückkehr aus einer Methode im Nichts landen und die Java-VM möglicherweise zum Absturz bringen). Die Wiederherstellung des Ausführungszustands ist also eine Funktion, die für einzelne Objekte nicht sinnvoll ist, sondern davon abhängt, dass dieselben Instanzen gleichermaßen wieder erreichbar sind, bevor die Threads wieder loslaufen.

Die technischen Details außer Acht lassend: Java wurde in erster Linie für die Erstellung von Applets entworfen. Mit der Zeit entwickelte sich Java sicherlich weiter, aber die Grundstruktur (die Virtuelle Maschine) ist im Prinzip noch gleich aufgebaut. Mobile Agenten waren im Java-Entwurf nicht geplant. Es ist also nachvollziehbar, dass frühe Java-Versionen keine starke Migration kennen, weil sie bei Applets nie benötigt wird. Dass auch spätere Versionen dies nicht vorsehen, liegt vermutlich daran, dass erhebliche Eingriffe in die Virtuelle Maschine vorgenommen werden müssten.

Das JESSICA2-Projekt (http://www.csis.hku.hk/~clwang/projects/JESSICA2.html) beschäftigt sich tatsächlich mit der Migration von Threads; dies wird über einen modifizierten JIT-Compiler auf Basis einer offenen Java-VM (Kaffe openVM) erreicht. Diese Java-VM wird auch als DJVM (Distributed Java VM) bezeichnet. Allerdings handelt es sich um passive Migration („[...] the application is unaware of the migration operation.“)[1].

Der Entwicklungsprozess im Java Community Process sollte an sich Hoffnung machen, dass auch Spezialanforderungen wie die starke Migration behandelt werden. Allerdings erfordert auch dieser Prozess eine Abstimmung, gerade bei Eingriffen in die innere Struktur der Java-VM. Solange man nicht eine größere Schar an Unterstützern aufweisen kann, wird auch der JCP keine Spezialwünsche akzeptieren.

[1] Zhu, Wang, Lau: Lightweight Transparent Java Thread Migration for Distributed JVM, ICPP2003

Aufgabe 2.4 – 15 Punkte

a) **Erwünschtes Verhalten:**

Der Agent migriert zu Beginn zunächst zur Stelle „MyHome“. Dort holt er sich eine Liste von Aufgaben ab. In dieser Liste gibt es eine Aufzählung von Reisezielen; diese besucht er der Reihe nach, schlägt jedesmal in seiner Aufgabenliste nach, was er am jeweiligen Ort zu tun hat, führt dies aus und migriert weiter. Sollte es zu einem Problem kommen, was durch den Rückgabewert ABORT repräsentiert wird, bricht er seine Reise ab.

Zum Schluss wandert er zur Stelle „MyHome“ zurück. Dort berichtet er in nicht genauer definierter Weise. Dann wird er wahrscheinlich beendet werden, da er die Methode „start“ verlässt, aber dies obliegt dem jeweiligen Agentensystem.

Tatsächliches Verhalten:

Leider funktioniert diese Implementierung auf diese Weise nicht, wenn man Standard-Java zugrunde legt. Denn durch die „migrate“-Anweisungen wird der Programmablauf unterbrochen, und da Java keine starke Migration unterstützt, beginnt der Agent seine Ausführung wahrscheinlich wieder von vorne. Er verliert dabei auch die Information, wie weit er in seiner Route gekommen ist. Das heißt konkret:

Zunächst stellt er fest, ob er auf „MyHome“ verweilt. Wenn nicht, migriert er dorthin und beginnt von vorne. Dies funktioniert wie erwartet, denn wir nehmen an, dass die Migration fehlerfrei funktioniert. Diesmal befindet er sich auf MyHome, sodass er sich die Aufgabeliste holt und daraus die Reiseroute gewinnt.

Ist die Liste nicht leer, betritt er die while-Schleife. Er holt sich die erste Stelle und migriert dorthin.

Aufgrund der schwachen Migration wird er nun die start-Methode von vorne beginnen. Dabei stellt er fest, dass er nicht auf „MyHome“ verweilt und migriert dorthin. Dadurch wiederholt er immer wieder diesen Ablauf, ohne Fortschritt zu erzielen – solange überhaupt die Aufgabenliste ein zweites Mal noch verfügbar ist.

Wäre die erste Stelle der Route „MyHome“, dann würde er sich dort nach Neustart die Aufgabeliste direkt erneut holen mit gleichem Effekt.

b) **Implementierung**

```
public class WanderAgent extends MobileAgent {

   Location startplace = ... ; // zeigt auf "MyHome"
   Itinerary m_it;
   TaskList m_Tasks;
   Location m_next;

   int m_nState = START;
   private final static int START=0;
   private final static int DONE=1;
   private final static int ATHOME=2;
   private final static int INLOOP=3;
   private final static int NEXTPLACE=4;
   private final static int AFTERLOOP=5;
   private final static int FINAL=6;

   public void start() {
      while (!m_nState==DONE) {
         switch (m_nState) {
            case START:
               if (!thisPlace().equals(startplace)) {
                  m_nState = ATHOME;
                  migrate(startplace);
               }
               else m_nState = ATHOME;
               break;
            case ATHOME:
               m_Tasks = thisPlace().getTaskList();
               m_it = m_Tasks.getItinerary();
               m_nState = INLOOP;
               break;
            case INLOOP:
               if (!m_it.hasNext()) m_nState = AFTERLOOP;
               else {
                   m_next = m_it.getNextPlace();
                   m_nState = NEXTPLACE;
                   migrate(m_next);
               }
               break;
            case NEXTPLACE:
               int state = m_tl.doTask(m_next);
               if (state==ABORT) m_nState = AFTERLOOP;
               else m_nState = INLOOP;
               break;
            case AFTERLOOP:
               m_nState = FINAL;
               migrate(startplace);
               break;    // kann weggelassen werden
            case FINAL:
               report();
               m_nState = DONE;
         }
      }
   }
}
```