Hauptseminar Wintersemester 1996/97

„Mobile Agenten”

Thema:

JAVA-TO-GO

ausgearbeitet von Antje Raap

Universität Stuttgart
Institut für Parallele und Verteilte Höchstleistungsrechner
Abteilung: Verteilte Systeme
Prof. Kurt Rothermel
Fritz Hohl

Inhaltsverzeichnis

Literatur

[1] Java-To-Go Itinerative Computing Using Java:
http://ptolemy.eecs.berkeley.edu/dgm/javatools/java-to-go

[2] Patrick Niemeyer, Joshua Peck: *Exploring Java*, O'Reilly & Associates Inc. (1996)

[3] Arthur von Hoff: *Hooked on Java*, Addison-Wesley Publishing Company (1996)

1 Einleitung

Java-To-Go ist eine experimentelle Infrastruktur für Mobile Agenten und agentenbasierende Applikationen, die von Weiyi Li von der Universität Berkley in Kalifornien entwickelt wurde. Sie beschäftigt sich mit „itinerative computing", diese „umherreisenden" Softwareagenten sind ein Satz von auf der Programmiersprache JAVA basierenden Applikationen.
Wie in anderen Mobilen Agentensystemen wird eine Anzahl von Rechnern von einem Mobilen Agenten oder parallel von einer Gruppe von Agenten durchreist.
Großer Wert wird bei Java-To-Go auf die sehr einfach einzurichtende Umgebung gelegt. Es wurde als Testumfeld zur Demonstration der Anwendbarkeit von agentenbasierenden Anrufsignalen in einem Telekommunikations Netzwerk erstellt.

2 Infrastruktur

Java-To-Go besitzt eine Infrastruktur, die anderern Agentensystemen stark ähnelt.
Es basiert auf zwei Komponenten:

- Der Mobile Agent, ein JAVA Programm, das von einem Rechner zum anderen wandert und alle Berechnungen und Verarbeitungen ausführt.
- Der Agentenserver ist der Teil des Agentensystemes, der sich auf dem jeweiligen von den Agenten durchlaufenen Rechnern befindet. Er wird bei Java-To-Go *Hall Server* genannt.

2.1 Der Mobile Agent

Der umherreisende Agent initiiert Berechnungen im Netzwerk von sich aus. Die Methodik steht im Gegensatz zum bekannten RPC(Remote Procedure Call)-Aufruf. Der Agent ist nicht beschränkt „Was" und „Wo" er arbeiten soll. Zuerst wird der Mobile Agent wie ein JAVA Applet, ein kleines Java-Programm, zu einem anderen Ort transportiert. Das Applet kann auf äußere Einflüße reagieren, z.B. von einem Browser aufgerufen werden. Es verhält sich passiv. Ein Mobiler Agent dagegen führt Aktionen oder Berechnungen aus, die von ihm selbst initiiert werden. Dieses Verhalten wird „proaktiv" genannt.

2.1.1 Komponenten

Zu jedem Agenten gehören drei Komponenten:

- die Klassendefinitionen, die wiederum aus mehreren Klassen bestehen können

- die Statusdefinitionen
- die Reiseroute, eine Liste der Hall Server, die der Agent besuchen soll

Die Klassendefinitionen können zwischen den Server nicht geändert werden. Im Gegensatz zu diesen statischen Klassendefinitionen sind die Statusdefinitionen und die Streckenwahl dynamisch.

2.1.2 Arbeitsschritte

Folgende Schritte durchläuft der Java-To-Go Agent auf einem Hall Server (Abbildung 1):

- Initalisierung: *boot*
  Wird als erste Funktion eines Agenten zur Aktivierung auf einem neuen Host aufgerufen.
  Die Werte der Berechnungen, die auf einem vorher besuchten Host erstellt wurden, werden den Statusvariablen zugewiesen.
- Hauptverarbeitung:
  Hier werden die lokalen Dienste des Hall Servers in Anspruch genommen. Außerdem kann die „Agent-zu-Agent"-Kommunikation durchführt werden.
- Aufräumprozedur: *wrapup*
  Wird am Ende, bevor der Agent den lokalen Hall Server verläßt aufgerufen.
  Diese Funktion sichert das Ergebnis der Berechnungen in die Statusvariablen im Agenten und gibt lokale Ressourcen frei. Der Agent leitet seine Übertragung auf dem folgenden Hall Server ein.

Java-To-Go Agenten haben *nicht* die Möglichkeit, ihre Arbeit auf einem Rechner zu suspendieren und auf einem nachfolgenden wiederaufzunehmen.

2.1.3 Oberfläche

Zwischen Agenten und Hall Server benötigt Java-To-Go eine definierte Schnittstelle. Die Agenten Schnittstelle ist allen Hall Servern bekannt, um mit den Agenten zu arbeiten, ohne die exakte Semantik zu kennen.

Ebenso hat der Hall Server eine lokale Schnittstelle definiert. Diese verbindet den Agenten mit den lokalen Ressourcen, aber isoliert dabei die Hall Server Implementierung. Diese Schnittstelle kann erweitert werden. Beispielsweise ist eine Schnittstelle zur Reservierung von lokalen Ressourcen denkbar, die auf den Agenten speziell zugeschnitten ist.

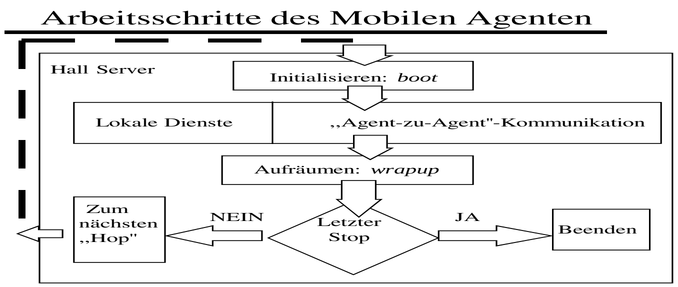

Abbildung 1: Arbeitsschritte des Mobilen Agenten

2.2 Interaktion zwischen Agent und Hall Server

Normalerweise besteht zwischen dem Agenten und seinem Herkunftsort keine Interaktion. Allerdings gibt es eine Ausnahme:
JAVA Objekte werden mit Hilfe ihrer Klassendefinitionen instantiiert. Benötigt der Agent weitere Objekte, deren Klassen nicht mit dem Agenten übertragen wurden, dann müssen Transaktionen angestoßen werden, um die Übertragung dieser Klassen zu initiieren. Erst wenn diese beendet ist, kann mit der Instantiierung dieser Klassen die Arbeit des Agenten fortgesetzt werden.
Diese Verfahrensweise verringert zwar die Gepäckmenge, die der Agent bei der Übertragung zu einem Hall Server bei sich trägt, aber es verkompliziert die benötigte Infrastruktur.

2.2.1 Klassenserver

Es muß an einer bekannten Stelle ein Klassen Server vorhanden sein, der den lokalen Hall Server mit den vom Agenten benötigten Klassendefinitionen versorgt. Wenn ein Agent instantiiert ist, weist der Hall Server ihm den zugehörigen Klassen Server zu. Dieser Klassen Server ist mit dem Wissen ausgerüstet, an welcher Stelle sich der Heimatklassen Server des Agenten befindet und welche Reiseroute der Agent bisher gewählt hat.

Empfängt der Hall Server eine Anfrage des Agenten über eine weiter benötigte Klasse, dann gibt der Hall Server diese Anfrage an den zugehörigen Klassen Server des Agenten weiter.
Voraussichtlich hat der Agent die Klasse auf dem Server bereits benutzt. Wenn sie allerdings nicht gefunden wird, dann wird auf dem Klassen Server des Heimatortes des Agenten gesucht. Dieser muß die Implementierung der Klasse besitzen. Die Klasse wird dann vom Klassen Server auf dem gerade vom Agenten besuchten Server transferiert und lokal instantiiert.

2.2.2 Datenübertragung über das Netz

Die jeweilige mit dem Agenten kommunizierende Maschine muß in der Regel am Netz und in Betrieb sein. Andere Hall Server die vorher oder später vom Agenten durchlaufen werden, müssen nicht verbunden sein.

Um die Übertragungszeit im Netz zu minimieren, wird nur der Kern des Agenten von einem Hall Server zum anderen übertragen. Andere Klassen werden nur, wenn sie benötigt werden transferiert, dabei findet eine „Zwei-Schritt-Suche" (siehe dazu 2.2.1.) nach den benötigten Klassen statt: erst lokal, dann entfernt.

2.3 Der Hall Server

Die Orte, die der Agent während seines Aufenthaltes im Netz besucht, sind die Hall Server.
Ein oder mehrere Hall Server können von einem Klassen Server versorgt werden. Der Klassenserver speichert die Klassen, die der Agent während des Besuches der zugehörigen Hall Server benötigt.

Der Kern des Hall Servers ist nur mit einer geringen Funktionalität ausgestattet. Dabei handelt sich sich um die kleinstmöglichste Funktionsanzahl, die die Mehrzahl der Agenten benötigt. Falls spezielle Funktionen ausgeführt werden sollen, können fast alle Teile des Hall Server bei der Implementierung angepaßt werden.

Hall Server sind ebenso wie Agenten in JAVA realisiert.

2.3.1 Aufgaben

Der Hall Server stellt für den Agenten lokale Dienste zur Verfügung. Er beeinflußt die Agenten Berechnungen nur passiv. Insbesondere ermöglicht der Hall Server die Weitergabe des Agenten zum nächsten gewünschten Knoten. Für die Kommunikation von einem Hall Server zum anderen wird eine TCP/IP-(Transmission Control Protocol/Internet Protocol)Verbindung benutzt.
Die agenten-regulierbare Streckenwahl durch das Netz wird vom Hall Server unterstützt.

3 Charakteristika

Das Anwendungsgebiet eines Agentensystems hängt entscheidend von den Eigenschaften des Systems ab, die wichtigsten werden in diesem Kapitel beschrieben.

3.1 Kommunikation

Die Agenten kommunizieren **lokal**, d.h. die kommunizierenden Agenten befinden sich auf demselben Hall Server. Dabei tauschen die Agenten Informationen über Nachrichten aus. Nachrichten werden in Java-To-Go als Klassen realisiert und benutzen die JAVA-Funktionalität.
Zur Identifikation des Kommunikationsteilnehmers muß der Name bekannt sein.

3.2 Namen und Adressen

3.2.1 Hall Server

Die Hall Server werden durch ein einmaliges **Alias** identifiziert.
Der Alias wiederum wird anhand des Hostnamen nach dem eindeutigen DNS (Domain-Name-System) und eine TCP(Transmission Control Protocol)-Portnummer festgelegt.
Dabei ist es möglich, mehrere Hall Server auf einem einzigen Rechner laufen zu lassen, wenn jeder Hall Server eine andere TCP-Portnummer benutzt.
Aufgrund des alias-basierenden Systems können Änderungen der Reiseroute leicht durchgeführt werden.

3.2.2 Agent

Der Name des Agentens ist eine Zeichenkette.
Es gibt kein vernünftiges Nameskonzept für Agenten, da diese bisher über das gesamte Agentensystem eindeutige Namen besitzen müssen.

3.3 Migration

Mit Java-To-Go wird eine **„halbe"** Migration durchgeführt. Programme und Daten migrieren. Der Zustand wird nicht auf einen anderen Rechner übernommen.
Andere Agentensysteme führen eine „transparente" Migration durch. Weili Li ist allerdings der Meinung, daß dies nicht für alle agentenbasierende Umgebungen, insbesondere für Java-To-Go notwendig ist([1]).
Die Daten werden mit Hilfe von der Methode „wrapup" (siehe dazu 2.1.2.) vor dem Rechnerwechsel im Agenten gesichert.
JAVA beinhaltet für den Transport über TCP/IP spezielle Methoden.

3.4 Sprachen

Die Implementierung des Agentensystems besteht aus einer Bibliothek von Klassen, die ebenso wie die Agenten ausschließlich in der objektorientierten Sprache JAVA programmiert sind.
Das Agentensystem benutzt das JDK (Java Development Kit) 1.0, bzw. für die Änderungen in der neusten Version 0.97 pre-alpha das JDK 1.0.2. Damit sollte es möglich sein, es unverändert auf allen JAVA-unterstützenden Plattformen auszuführen, vorausgesetzt diese halten sich an die von Sun Microsystems gesetzten JVM(Java Virtual Machine)-Spezifikationen.

JAVA hat zum Ziel, eine nahtlose Netzkommunikation und Sicherheitsmanagement zu realisieren. Der Anwendungsbereich liegt innerhalb der Internet-Programmierung, bei der heterogene Rechner über das einheitliche TCP/IP kommunizieren.

In Bezug auf die Syntax ähnelt JAVA der Programmiersprache C bzw. C++. Es kann mit den Eigenschaften schnell, verläßlich und rechnerunabhängig charakterisiert werden. Die Programmiersprache ist einfach. Sie beinhaltet weniger komplizierte Konstrukte, wie etwa C++: weder Overloading, Zeigerarithmetik, noch implizite Typenkonvertierung.
Zusätzlich wird automatisches Garbage Collection durchgeführt. Java ist „Multi- threaded", d.h. es können mehrere Anfragen zur gleichen Zeit ausgeführt werden.

Im Gegensatz zu CGI(Common Gateway Interface)-Programmen wird JAVA auf der Client-Seite ausgeführt. Wie C/C++ muß es nicht auf jedem Rechner kompiliert werden, sondern JAVA-Programme werden einmalig zu einfachen Bytecode kompiliert und versendet. JAVA-Programme können auf dem empfangenen Rechner systemunabhängig ablaufen. Dies war mit den traditionellen Programmiersprachen nicht möglich, da das kompilierte Programm nur auf einer bestimmten Plattform ausgeführt werden konnte. Ist das JAVA-Programm übertragen, ist es unabhängig von der Bandbreite und der Modemgeschwindigkeit.

Das Agentensystem ist auf JAVA zugeschnitten, damit müssen keine Konvertierungen

vorgenommen werden. Der gleiche Interpreter und Compiler kann einheitlich für das ganze System eingesetzt werden.
Da die Java-To-Go Agenten vollständig in JAVA geschrieben sind, können die Mobilen Agenten mit JAVA beliebig erweitert werden. Beispielsweise kann die Möglichkeit einer Erweiterung mit GUI (Graphical User Interface) genutzt werden.

3.5 Sicherheit

Auf Netzwerkebene wird mit TCP/IP keine Sicherheit im Transport gewährt.

Die Sicherheit bei der Kommunikation von Java-To-Go-Agenten mit den durchlaufenen Rechner wird auf eine streng definierte Schnittstelle eingeschränkt und durch die Eigenschaft des JDK gewährleistet. JAVA beinhaltet sowohl mehrere Mechanismen zur Vermeidung eigener Fehler, als auch gegen Angriffe von außerhalb. Die Robustheit gegen eigene Fehler wird durch das Design der Programmiersprache mit wenig komplizierten Sprachkonstrukten (keine Zeigerarithmetik, statische Typisierung, kein Overloading usw.) und dem Adress-Design mit automatischen Garbage Collection und expliziter Speicherallokierung gefördert. Gegen den Angriff von außen steht das dreistufige Sicherheitskonzept der Systemimplementierung (Abbildung 2):

- Der Verifikator garantiert die Integrität der ankommenden Klassen. Es wird geprüft, ob sich diese an die Basisregeln halten und ob die Typen der Instruktionen übereinstimmen.
- Der Klassenlader holt alle benötigten Klassen vom Netzwerk. Außerdem kontrolliert er, daß diese den JAVA Konventionen entsprechen.
- Der Sicherheitsmanager kontrolliert und schränkt ggf. den Zugriff auf Systemressourcen, Netzwerkkomponenten und Oberflächen ein.

Außer dem JAVA-Sicherheitskonzept gibt es bisher keine weiteren Sicherheitskonzepte in Java-To-Go. Insbesondere bei der „Host-zu-Host"- und „Agent-zu-Agent"-Kommunikation, die anhand des Hostalias bzw. Agentennamen durchgeführt wird, wären ein sicheres Authentifikationsverfahren wünschenswert.

3.6 Dienste und Trading

Java-To-Go sieht **keine** besonderen Komponenten als Trader im System vor.
Es gibt keinen speziellen Typ für Agenten, die Dienste ausführen.

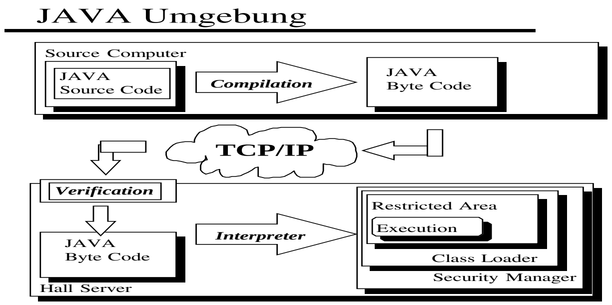

Abbildung 2: JAVA Umgebung

3.7 Fehlerbehandlung

Es gibt keine spezielle Unterstützung bei Systemabstürzen. Die Agenten und Nachrichten sind weder persistent, noch ist die Verarbeitung transaktiosorientiert.

Von Seiten der Programmiersprache JAVA steht ein spezielles Sprachkonstrukt (try, catch) zur Behandlung von Fehlern zur Verfügung. Das JDK beinhaltet einen Debugger.
Zur Fehlersuche stellt Java-To-Go ein Klasse „java2go.debug" zur Verfügung, die sowohl bei textueller, als auch graphischer Oberfläche verwendet werden kann.
Ein GUI-basierter Agentenverfolger ist geplant.
Da Agenten von außen nicht terminiert werden können, besteht bei außer Kontrolle geratenen Agenten nur die Möglichkeit, den gerade besuchten Hall Server zu stoppen.

3.8 Interaktion mit dem Benutzer

Die Server können sowohl mit graphischer als auch mit textueller Oberfläche gestartet werden. Die Server können als eigenständige Umgebung oder in einem Browser, der das JAVA Laufzeitsystem enthält, ablaufen.
Es gibt bisher keine Benutzungsoberfläche als Hilfsmittel für die Erstellung, Steuerung von Agenten und Planung von Agentenabläufen. Dringend nötig ist der geplante GUI-basierte Agentenverfolger.

4 Installation

Java-To-Go ist ein Toolkit, das eine experimentelle Struktur bietet. Es besteht aus einer Bibliothek von Klassen, die die Funktionalität ermöglichen, eine agenten-fähige Umgebung zu implementieren. Die Installation ist einfach.
Die derzeitige Version 0.97 pre-alpha unterscheidet sich nicht signifikant von den Vorgängern 0.95 und 0.96.
Es gibt vier Installations-Voraussetzungen:

- Betriebssystem(Hardware)
  Java-To-Go wurde bereits auf drei Plattformen getestet und erfolgreich eingesetzt:
  - Solaris 2.4 auf Sun SparcStation 2 und 20
  - Linux 1.2.13 für Java-To-Go 0.96 auf Intel i486
  - Microsoft Windows 95 auf Intel Pentium
- Es müssen Portierungen für das Sun Microsystem JDK Version 1.0.2 (für Linux nur JDK 1.0) zur Verfügung stehen.
- Es werden TCP-IP Protokoll Stacks benötigt.
- Die Java-To-Go Quelltexte und Systemklassen sind in verschiedenen Formaten verfügbar. Dafür wird ein Entarchivierungsfunktion benötigt, z.B. das UNIX-Kommando „tar". Für die mitgelieferten Quelltexte und den Bytecode sollte ein Plattenplatz von etwa 1 MByte bereitstehen.

Die zur Verfügung stehenden Systemdokumentation ist verständlich, gehen allerdings nicht in die Tiefe. Als Anwendungsbeispiel dient ein Verzeichnis von drei sehr einfachen Agenten und dem Agentenserver.
Java-To-Go steht kostenlos zur Verfügung und kann einfach heruntergeladen werden. Es besteht allerdings kein Anspruch auf Software-Service. Der Verfasser bietet Interessenten einen Email-Kontakt an.

5 Beispielanwendung

Im Rahmen des Hauptseminares sollte das Agentensystem installiert werden und eine Agentenanwendung beispielhaft programmiert und vorgeführt werden.

5.1 Aufgabenstellung

Die Aufgabenstellung mußte für das Java-To-Go-System leicht abgeändert werden, da es mit lokaler Kommunikation und ohne Trading-Komponente arbeitet.
Die Agenten werden mit einem „Launcher" aufgerufen. Dieses Programm initialisiert den Agenten mit den benötigten Werten und startet den Agenten auf dem ersten Hall Server der Reiseroute, dem Host „beta".
Beispielanwendung:

- DateiSuchAgent
  1. Der Agent migriert zu dem Host „alpha",
  2. wartet auf den DateiServerAgent und übergibt diesem den gewünschten Dateinamen,
  3. wartet auf die Antwort von dem DateiServerAgent und übernimmt die angeforderte Datei (Abbildung 3),
  4. migriert mit Antwort zurück zu Heimathost „beta"
  5. und zeigt dem Benutzer den Inhalt der Datei an.
- DateiServerAgent
  1. Der Agent migriert zu dem Host „alpha",
  2. wartet auf den DateiSuchAgent und erhält von diesem den Dateinamen,
  3. sammelt die Datei auf
  4. und händigt dem DateiSuchAgenten die Datei aus (Abbildung 3).

5.2 Erfahrungen und Zeitmessung

Als Beispiel für einen Agenten stand ein in der Dokumentation beschriebener Agent zur Verfügung. Dieser befand sich jedoch nicht in der Installationsdatei, wurde mir aber, auf Anfrage per Email von Weiyi Li zugeschickt.
Da Java-To-Go komplexe Klassen zur Verfügung stellt, ist der Quellcode der Beispielagenten kurz und übersichtlich (siehe dazu Anhang 8.).
Mit Hilfe von JAVA-Kenntnissen ist es möglich, das beschriebene Beispiel zu programmieren und mit Hilfe der GUI-basierenden Server vorzuführen.

Beispielagent

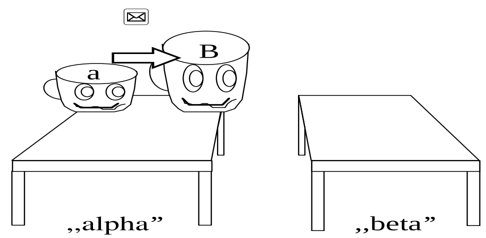

Abbildung 3: DateiServerAgent „a" händigt dem DateiSuchAgenten „B" die Datei auf dem Hall Server „alpha" aus.

Für die Zeitmessungen wurde die JAVA Systemklasse „System.currentTimeMillis()" benutzt. Die Zeitmessung ergab sich aus fünf Testläufen.

Migrationzeit eines Agenten: 0,204 bis 0,225 Sekunden
Zeit einer einzelnen Nachricht: 0,017 Sekunden
Gesamtzeit der Agentenkommunikation: 0,067 bis 0,073 Sekunden
(inklusiv des Dateieinlesen durch den DateiServerAgenten)
Gesamtzeit des DateiSuchAgenten: 0,522 bis 0,541 Sekunden

6 Zusammenfassung

Java-To-Go ist kein kommerzielles, ausgereiftes oder fehlerfreies Softwareprodukt.
Im Gegensatz zu anderen mobilen-agenten-Systemen soll Java-To-Go in erster Linie zu experimentellen Zwecken dienen und bietet nur einen minimalen Funktionsumfang.

Sieht man das Agentensystem unter diesem Aspekt, so ist die einfache Installation mit wenig Mitteln, die erweiterbare, offene Umgebung und insbesondere der Vorteil der eingesetzten Sprache mit ihrem Sicherheitskonzept hervorzuheben.

7 Ausblick

Einige grundlegende Konzepte und Probleme eines Agentensystems sind in Java-To-Go noch nicht gelöst. Insbesondere sollten folgende Punkte erarbeitet werden:

- Ein zentrale Koordinationsmöglichkeit, um einen Systemüberblick zu erhalten und insbesondere fehlerhafte Agenten zu stoppen, ohne den Server zu beenden.
- Neben dem Sicherheitskonzept von JAVA ist ein Konzept in den Bereichen „Host-zu-Host"- und „Agent-zu-Agent"-Kommunikation vonnöten.
- Die Entwicklung eines systemweiten Namenskonzept für Agenten wäre vorteilhaft.
- Mit dem geplanten Agentenverfolger würde die Interaktion zum Benutzer gefördert werden. Außerdem wäre eine Benutzungsoberfläche zur Erstellung und Steuerung von Agenten, sowie zur Planung von Agentenabläufen vorstellbar.

Weitere vergleichbare Agentensysteme werden im Rahmen dieses Hauptseminares vorgestellt. Insbesondere Cyber-Agents und Mole, die ebenfalls mit JAVA arbeiten, sind hervorzuheben.

8 Anhang: Quelltext der Beispielanwendung

8.1 DateiSuchAgent

```
/**************************************************************************

  Beispielagent fuer das Agentensystem JAVA-TO-GO im Rahmen des
  Hauptseminares "Mobile Agenten" WS 96/97.

  Agentensystem: JAVA-TO-GO Version 0.97 pre-alpha
  Programmiersprache : Java Version 1.0.2
  Programm-Id:   DateiSuchAgent.java
  Version:       1
  Programmierer: Antje Raap
  Datum:         15.01.97

  Vor dem Start der Agenten muessen die Server "alpha" und "beta" gestartet
  werden.
  Die Agenten werden mit einem "Launcher" aufgerufen.
  Dieses Programm initialisiert den Agenten mit den benoetigten Werten:
  Objektname:   "Charlie Brown"
  Klassenname:  "DateiSuchAgent"
  Heimatserver: "beta"
  Reiseroute:   "beta;alpha;beta"
  und startet den Agenten auf dem ersten Hall Server der Reiseroute, dem Host
  "beta".

**************************************************************************/

import java2go.basics.*;
import java2go.modules.*;
import java2go.util.*;
import java.io.*;
import java.util.*;

//
// Agents implemented by this class hands off the filename to
// an other agent and retrieves the desired information fetched
// by the other agent back to the home server.
//
// An example of an info-fetching agent is implemented in the
// 'DateiServerAgent.class'
```

```
//

public class DateiSuchAgent extends Template implements Agent {

    StringBuffer resultString = null;

    // constructor...
    //  allocate enough space for state during execution;
    //  doesn't have to be here, though
    public DateiSuchAgent () { resultString = new StringBuffer(); }

   // INITIALIZATION ---------------------------------------------

    // bootstraps the values for the agent to use...
    protected void boot(byte[] initValues) {

        if (initValues != null) {
            DataInputStream din = new
                DataInputStream(new ByteArrayInputStream(initValues));

            try {
                resultString.append(din.readUTF());
            } catch (IOException e) {}
        }
    }

   // FINALIZATION ----------------------------------------------

    // saves the values so that the next bootstrap can be performed
    // at the next host
    protected byte[] wrapup() {

        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        DataOutputStream dout = new DataOutputStream(bout);

        try {
            dout.writeUTF(resultString.toString());
        } catch (IOException e) {}

        return bout.toByteArray();
    }

   // ACTIVATION CRITERIA (for the agent server) ---------------------
```



```
    // do we care if there is a need for an unplanned,
    // intermediate forwarding node?
    public boolean exactPath() { return false; }

   // AGENT BEHAVIOR -----------------------------------------------

    // go to the next host
    private void goToNextHost() {

        // obtain the bootstrap data from this computation
        byte[] result = wrapup();

        // This loop is used to attempt use-controlled reconnection
        //      agent can wonder to over an alternate route if the
        //      connection isn't up correctly.
        while (true) {

            try {

                // going to the designated host
                local.go(this,result);

                break;

            } catch (IOException e) {

                local.println("Problem with connection. ");
                local.println("    ("+e.getMessage()+")");
                local.println(" Waiting...");
            }

            // wait for a second before making another attempt
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }

    // body of the agent
    public void run() {

        // message to hall server
```

```
local.println("I am now working on "+local.host()+"!");

// where (what) should I go (do) next?
if (local.endOfList()) {

   // show dataset contents on host and system
   local.println("Contents of the file:");
   local.println(resultString.toString());
   System.out.println("\n"+resultString.toString());
   local.println("Journey ended. Shutting down!");

} else {

  if (local.host().equals ("alpha") ) {

     // get a communication handle...
     CommHandle chHandle = getCommHandle("getDataset");

     try {

        Message mesg =
            new Message("Peppermint Patty",
                        (Object) "/home/raapae/Letter"
                       );
        local.println("Waiting for communication.");

        while (true) {
          try {
               // Give the info to the second agent
              chHandle.write(mesg);
              break;
          } catch (IOException e) {}

          try {
               Thread.sleep(100);
          } catch (InterruptedException e) {}
        }

        // waiting to receive dataset contents from DateiServerAgent
        local.println("Waiting for the desired file.");
        resultString.append(chHandle.read().content());

        chHandle.releaseHandle();
```



```
            } catch (IOException e) { local.println("IO Exception!"); }

         }
         // else-clause: do nothing special

         // pack up and go
         goToNextHost();

       }

       // exits stage left
       local.goodbye();
    }
}
```

8.2 DateiServerAgent

```
/***************************************************************************

  Beispielagent fuer das Agentensystem JAVA-TO-GO im Rahmen des
  Hauptseminares "Mobile Agenten" WS 96/97.

  Agentensystem: JAVA-TO-GO Version 0.97 pre-alpha
  Programmiersprache : Java Version 1.0.2
  Programm-Id:   DateiServerAgent.java
  Version:       1
  Programmierer: Antje Raap
  Datum:         15.01.97

  Vor dem Start der Agenten muessen die Server "alpha" und "beta" gestartet
  werden.
  Die Agenten werden mit einem "Launcher" aufgerufen.
  Dieses Programm initialisiert den Agenten mit den benoetigten Werten:
  Objektname:   "Peppermint Patty"
  Klassenname:  "DateiServerAgent"
  Heimatserver: "beta"
  Reiseroute:   "beta;alpha" oder
                beliebig "gamma;theta;gamma;theta;alpha"(Server muessen laufen)
  und startet den Agenten auf dem ersten Hall Server der Reiseroute, dem Host
  "beta".

***************************************************************************/
```

```
import java2go.basics.*;
import java2go.modules.*;
import java2go.util.*;
import java.io.*;
import java.util.*;

//
// This class implements agents travels to a list of
// destinations (set by the agent launcher), until the host at
// the end of this list is encountered. There this agent waits
// to get the name of the file, he is going to collect. After
// collection, he transfers its contents to the other agent.
//
// An example of an info-asking agent is implemented in the
// 'DateiSuchAgent.class'
//

public class DateiServerAgent extends Template implements Agent {

    StringBuffer resultString = null;

    // constructor...
    //  allocate enough space for state during execution;
    //  doesn't have to be here, though
    public DateiServerAgent () {
        resultString = new StringBuffer();
    }

   // INITIALIZATION ---------------------------------------------

   // bootstraps the values for the agent to use...
    protected void boot(byte[] initValues) {

        if (initValues != null) {
                DataInputStream din = new
                DataInputStream(new ByteArrayInputStream(initValues));

            try {
                resultString.append(din.readUTF());
            } catch (IOException e) {}
        }
    }
```



```
// FINALIZATION ---------------------------------------------

 // saves the values so that the next bootstrap can be performed
 // at the next host
 protected byte[] wrapup() {

     ByteArrayOutputStream bout = new ByteArrayOutputStream();
     DataOutputStream dout = new DataOutputStream(bout);

     try {
         dout.writeUTF(resultString.toString());
     } catch (IOException e) {}

     return bout.toByteArray();
 }

// ACTIVATION CRITERIA (for the agent server) ---------------------

 // do we care if there is a need for an unplanned,
 // intermediate forwarding node?
 public boolean exactPath() {

     // in this case, just skip to the next node
     return false;
 }

// AGENT BEHAVIOR ---------------------------------------------

 // go to the next host
 private void goToNextHost() {

     // obtain the bootstrap data from this computation
     byte[] result = wrapup();

     // This loop is used to attempt use-controlled reconnection
     //      agent can wonder to over an alternate route if the
     //      connection isn't up correctly.
     while (true) {

         try {

             // going to the designated host
```

```
            local.go(this,result);

            break;

        } catch (IOException e) {

            local.println("Problem with connection. ");
            local.println("    ("+e.getMessage()+")");
            local.println(" Waiting...");
        }

        // wait for a second before making another attempt
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
    }
}

// body of the agent
public void run() {

    // message to hall server
    local.println("I am now working on "+local.host()+"!");

    // where (what) should I go (do) next?
    if (local.endOfList()) {

        // get a communication handle...
        CommHandle chHandle = getCommHandle("getDataset");

        local.println("Waiting for communication.");
        local.println(resultString.toString());

        try {
            // At the last station, I'll WAIT until
            //      I can obtain an ok from home (another agent)

            // get the dataset name
            Message newMessage = chHandle.read();
            local.println("Received Filename:"+
                          ((String) newMessage.content())
                         );
            try {
```



```
                // get dataset contents and save it in resultString
                FileInputStream fin = new FileInputStream(
                                      (String) newMessage.content());
                DataInputStream din = new DataInputStream(fin);
                do {
                  resultString.append(din.readLine()+"\n");
                } while (fin.available() > 0);
                fin.close();

                // if there is no file with the name, give a note
                } catch (FileNotFoundException eee) {
                    resultString.append("NO FILE WITH THIS NAME FOUND!\n");
                    local.println( (String) resultString.toString() );
              } catch (IOException ee) {}

              // transfer content of file to Charlie
              chHandle.write( new Message("Charlie Brown",
                             (Object) resultString.toString())
                            );

        } catch (IOException e) { local.println("IO Exception!"); }

        // release the communication handle
        chHandle.releaseHandle();
        local.println("Info transferred. Shutting  d o w n !     ");

    } else {

            // do nothing special,
            // pack up and go
            goToNextHost();
    }

    // exits stage left
    local.goodbye();
  }
}
```