

Universität des Saarlandes

**Ausarbeitung**  
**Seminar “AI Tools“**

Wintersemester 2006/2007

# **JADE und FIPA**

von

**Cathrin Weiß**

(cathrin.weiss@gmail.com)

2031382

**Betreuer**

Dr. Michael Kipp

## **Zusammenfassung**

Ein Multiagentensystem zu entwickeln ist eine herausfordernde Aufgabe, insbesondere hinsichtlich Interoperabilität mit anderen Systemen. Eine Möglichkeit, diese und weitere Probleme zu umgehen und den Implementierungsaufwand gering zu halten, ist JADE. JADE ist ein Java Framework zum Erstellen und Ausführen von Multiagentensystemen. Die Middleware verfolgt den FIPA Standard, welcher eine Norm für Kommunikation und Architektur von Agenten ist. Die nachfolgende Arbeit gibt eine Einführung in FIPA sowie JADE und seine Anwendungsmöglichkeiten.

# **Inhaltsverzeichnis**

<b>1</b>	<b>Einleitung</b>	<b>4</b>
<b>2</b>	<b>Multiagentensysteme nach FIPA</b>	<b>5</b>
<b>3</b>	<b>JADE</b>	<b>8</b>
<b>4</b>	<b>Zusammenfassung und Diskussion</b>	<b>15</b>

# 1 Einleitung

Multiagentensysteme haben in der Künstlichen Intelligenz eine hohe Bedeutung. Viele alltäglich benutzte Anwendungen verwenden mehr oder weniger offensichtlich solche Systeme, die dazu dienen, gewisse Prozesse nach bestimmten Regeln automatisch ablaufen zu lassen. Diese Systeme folgen dem Peer-to-Peer-Prinzip, das heißt, die Agenten sind in der Lage unabhängig von einer Serverinstanz miteinander zu kommunizieren. Sie zeichnen sich durch die Möglichkeit des asynchronen Nachrichten Transfers aus, d.h. ein Agent Bob kann einem Agenten Jon jederzeit eine Nachricht senden, unabhängig davon, ob Job nun gerade online im System ist. Betritt Jon das System, so ist er in der Lage, die Nachricht von Bob zu lesen. Um deutlich zu machen, wie ein solches Multiagentensystem funktioniert, betrachten wir uns ein einführendes Beispiel: einen elektronischen Buchladen. In diesem System gibt es Instanzen (Agenten), welche Bücher anbieten und Instanzen (Agenten), welche Bücher suchen und entsprechend kaufen möchten. Da ein Buch von mehreren Verkäufern zu unterschiedlichen Preisen angeboten werden kann, ist es natürlich für einen Interessenten wünschenswert, dass auch der günstigste Preis gefunden werden kann. Also ist es für den Käufer wichtig, dass er eine Anfrage absenden kann, die auch alle Anbieter erreicht. Dazu muss er wissen, welche Agenten überhaupt Anbieter sind. Danach muss er entscheiden können, welcher Anbieter das beste Angebot geschickt hat, um dann dort zu kaufen. Also sehen wir uns hier mit einem System konfrontiert, das bereits einige Anforderungen an die Agenten stellt.

Es liegt auf der Hand, dass Verhandlungen, Informationsaustausch und -verarbeitung nur funktionieren können, wenn eine Sprache verwendet wird, die von allen Agenten verstanden werden kann. Ist dies nicht der Fall, kann es zu Komplikationen, Fehlinterpretationen oder sogar einem kompletten Ausfall des Systems führen. Somit ist ein Standard erforderlich. Für Agenten gibt es den FIPA-Standard. Der FIPA-Standard, welcher seit 2005 IEEE Standard ist, beinhaltet Festlegungen für Agentenarchitektur, Sprache sowie Nachrichtenformat und -austausch in Agentensystemen.

Es wäre also wünschenswert, eine komfortable Möglichkeit zu haben, ein Agen-

tensystem direkt nach FIPA-Standard zu entwickeln ohne sich nun um die genauen Spezifikationen kümmern zu müssen. Genau diese Möglichkeit bietet JADE, eine Java Middleware.

Diese Arbeit befasst sich also mit dem FIPA-Standard und der Middleware JADE und wird anhand einfacher Beispiele zeigen, wie man Multiagentensysteme mit Hilfe von JADE FIPA-konform entwerfen, entwickeln und betreiben kann. Zunächst wird erläutert werden, wie ein Multiagentensystem funktioniert und wie es durch FIPA standardisiert wird. Im Anschluss daran wird die Anwendung von JADE anhand einfacher Beispiele demonstriert.

## **2 Multiagentensysteme nach FIPA**

Um die Arbeitsweise der Middleware JADE nachvollziehen zu können, ist es wichtig, einige Begriffe im Vorfeld zu erklären und Funktionsweisen zu erläutern.

FIPA ist eine IEEE-Standard Organisation, die sich zum Ziel gesetzt hat, agentenbasierte Technologie und die Interoperabilität ihrer Standards mit anderen Technologien zu normieren. Die FIPA Spezifikation stellt eine Sammlung von Standards dar, welche die Zusammenarbeit von Agenten sowie die Form ihrer Dienste festlegt. Die Spezifikation umfasst die Sprache der Agenten (ACL - Agent Communication Language), die Agentenarchitektur, den Nachrichtentransfer, sowie auch die Form der Nachrichten (Protokolle). Ich werde nachfolgend näher auf die konkreten Spezifikationen eingehen. Ein Multiagentensystem[4] ist ein komplexe Kombination verschiedener Bausteine. Eine Basiseinheit bilden die Agenten, über welche die Systemkommunikation läuft. Ein Agent[4],[5] ist nun also ein Programm, welches folgende Eigenschaften besitzt:

- Autonomie: ein Agent kann selbständig agieren.
- Proaktivität: ein Agent kann eine Aktion starten.
- Reaktivität: ein Agent kann auf eine Aktion eines anderen Agenten reagieren.

- Zielbasiertheit: das Verhalten eines Agenten zielt auf einen bestimmten Endzustand (Ziel), was wiederum seine Aktionen bestimmt.
- Soziales Verhalten: ein Agent ist in der Lage, mit anderen Agenten zu kooperieren, sofern dies dem Erreichen seines Ziels förderlich ist.
- Adaptivität: ein Agent kann sich einer veränderten Situation anpassen und lernen.

Die Interaktion, Kommunikation und auch die Architektur der Agenten muss derart genormt sein, dass ein Verständnis untereinander möglich ist. Dies ermöglicht der FIPA Standard. Wie wird nun also ein Agentensystem (AS) nach FIPA definiert? FIPA gibt ein so genanntes Referenzmodell vor.

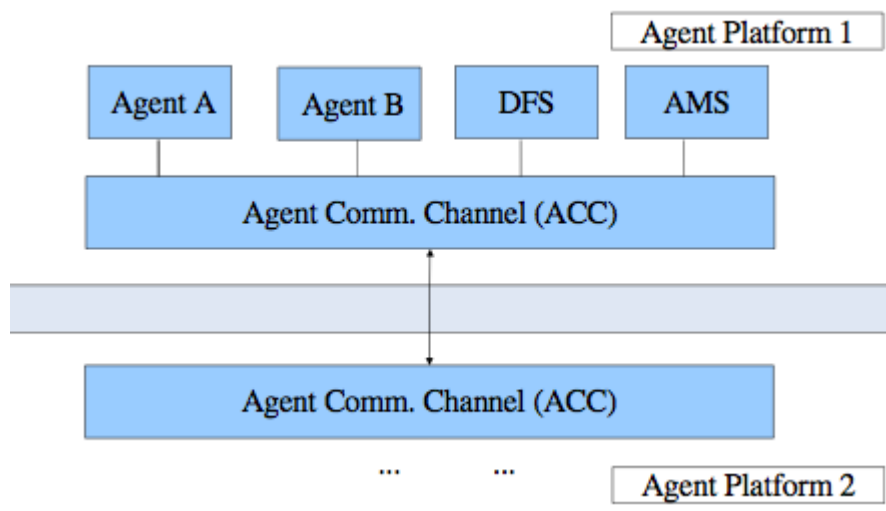


Abbildung 1: FIPA Referenzmodell

Die einzelnen Bestandteile:

- Agentenplattform: Die Agentenplattform ist die physikalische Infrastruktur (beispielsweise der Rechner selbst) und enthält Agenten und Steuereinheiten. Agentenplattformen können auch untereinander kommunizieren mit Hilfe des Kommunikationskanals, des ACC (Agent Communication Channel), welcher Austausch und Propagierung von Nachrichten regelt. An den ACC angehängt sind folgende Komponenten:

- Agent: Der Agent ist der Akteur im System und hat eine eindeutige ID. Er kann Dienste anbieten oder anfordern.
- AMS (Agent Management System): Das AMS ist die Kontroll- und Verwaltungseinheit, die für Erzeugung bzw. Löschung von Agenten im System zuständig ist. Man nennt das AMS auch “White Pages“. Es werden Funktionen wie `create()` (Agent anlegen), `execute()` (Aktion ausführen), `suspend()` (Agent temporär anhalten) und `terminate()` (Agent aus dem System entfernen) unterstützt.
- DFS (Directory Facility System): Das DFS registriert die Agenten im System. Es registriert sowohl die Agenten IDs, als auch jeweils angebotene Dienste. Suchanfragen von Agenten laufen ebenfalls über das DFS.

Nun ist die Agentenkommunikation zu betrachten. FIPA hat dafür eine Sprachspezifikation, die ACL (Agent Communication Language). Man kann sich die ACL Kommunikation ähnlich einer eMail Korrespondenz vorstellen. Auch eine eMail hat diverse Parameter, wie etwa Sender, Empfänger, Betreff oder Inhalt. Wichtige ACL-Parameter sind:

- `sender`: Der Agent, der eine Nachricht verschickt (Absender).
- `receiver`: Der Agent, an den die Nachricht gerichtet ist (Empfänger).
- `content`: Der eigentliche Nachrichteninhalt.
- `performative`: Die Intention der Nachricht. (Welches Anliegen hat der Sender an den Empfänger?) Es gibt einige festdefinierte Performativen im FIPA Standard, wie etwa
  - `CFP` (Call for proposals): Ein Agent möchte von anderen Agenten Angebote erhalten.
  - `PROPOSE`: Ein Agent unterbreitet ein Angebot.
  - `REFUSE`: Ein Agent kann kein Angebot auf Anfrage hin unterbreiten.
  - `ACCEPT PROPOSAL`: Ein Agent nimmt ein Angebot an.

- **INFORM**: Eine Information ohne weitere Kommunikationskonsequenzen wird ans System abgesetzt.
- **FAILURE**: Ein Vorgang ist fehlgeschlagen oder eine Anfrage kann nicht erfüllt werden.

Durch die Form einer Nachricht kann also eindeutig festgelegt werden, welches Anliegen ein Agent an einen anderen (oder auch gegebenenfalls an alle anderen - Angebot von Service) hat bzw. welche Aktion ausgeführt wird. Dadurch, dass die IDs eindeutig sind, ist es auch möglich, asynchronen Nachrichtentransfer zu betreiben. Es ist also möglich, dass ein Agent Bob einen zur Zeit gar nicht im System befindlichen Agenten Jon kontaktieren kann. Die Nachricht wird vom System genau dann an Jon überliefert, wenn dieser dem System wieder beitrifft. Kommunikationsbasis ist häufig ein *Service*, der von einem Agenten angeboten und von einem anderen Agenten gewünscht wird. Ein Service ist als Dienst aufzufassen, wie etwa Buchhandel-Service (ein Agent bietet den Dienst an und verkauft Bücher, ein andere möchte ein Buch kaufen und nutzt somit den Dienst). Eine komfortable Möglichkeit, Multiagentensysteme basierend auf Peer-to-peer Kommunikation nach FIPA Standard zu entwickeln und auszuführen, ist die Java Middleware JADE.

### 3 JADE

JADE[1] wurde von TILAB<sup>1</sup> entwickelt. Die Implementierung ist in Java, was eine weitgehende Plattformunabhängigkeit garantiert und somit den Einsatz der Systeme auf verschiedensten Hardwarearchitekturen ermöglicht - angefangen beim normalen PC oder Server über PDAs bis hin zu Handys. Diese Flexibilität lässt viel Freiraum für mannigfaltige Ideen. Folgende Philosophien werden verfolgt:

- **Interoperabilität**: JADE ist kompatibel zum FIPA Standard, so dass mit JADE entwickelte Systeme und Agenten mit jedem anderen FIPA konformen System kommunizieren können.

---

<sup>1</sup>Telecom Italia Lab

- Uniformität und Portabilität: JADE stellt eine homogene API zur Verfügung, welche unabhängig von Java Version und zu Grunde liegendem Netzwerk ist. Somit ist es dem Entwickler möglich, die Java Laufzeitumgebung dynamisch zu wählen.
- Anwendungsfreundlichkeit: Der Entwickler wird nicht mit der Komplexität der Middleware konfrontiert. Er nutzt einfach die von der API zur Verfügung gestellten Schnittstellen. Er ist auch nicht gezwungen alle Möglichkeiten von JADE zu kennen. Er braucht nur jene Teile zu implementieren, die er auch zwingend für die jeweilige Anwendung benötigt.

JADE ermöglicht auch die Einbindung weiterer Technologien und Sprachen. Beispielsweise ist eine Integration und Verwendung von "Jess", einem Regelsystem möglich. Dabei wird die Basisanwendung mit Hilfe vom JADE Framework in Java implementiert und die spätere Versorgung mit Regeln kann in Jess erfolgen. Es ist allerdings auch möglich, innerhalb des Java Codes mit Jess zu arbeiten, je nach Bedarf. Betrachten wir uns zunächst mal den Lebenszyklus eines JADE Agenten:

Jeder Agent wird mit der Methode `setup()` initialisiert. Das Verhalten eines Agenten wird mit so genannten behaviours bestimmt. Ein behaviour muss aktiviert werden z.B. durch einen Trigger, der aus der Umwelt bzw. von einem anderen Agenten kommt. Wird ein behaviour ausgeführt, so ruft dieses die Funktion `action()` auf. Ist ein behaviour beendet, so kann sie - je nach Zustand - beendet und somit deaktiviert werden (in diesem Falle wird die Methode `done()` ausgeführt) oder das behaviour hat noch weitere Aufgaben - sie bleibt im aktiven Zustand und wird weiter ausgeführt.

Behaviours können entweder

- genau einmal feuern und sich dann beenden oder
- sie können zyklisch sein.

Ruft ein Agent die Methode `doDelete()` auf, so verlässt er das System, welches ihn dann mit einen Aufruf von `takeDown()` entfernt. (Vgl. auch [2])

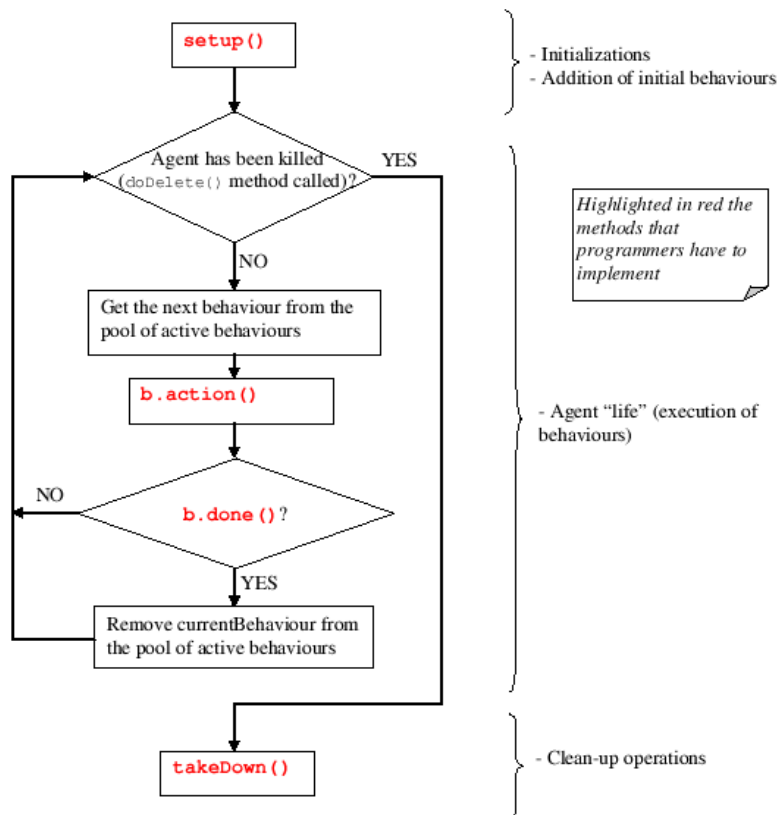


Figure 2. Agent Thread path of execution

Abbildung 2: Lebenszyklus eines Agenten

Nun möchte ich anhand eines konkreten Beispiels die Entwicklung eines Agentensystems mit JADE demonstrieren. Zunächst ein einfaches HalloWelt Programm in Abb. 3.

In diesem einfachen Beispiel wir einfach nur ein Hallo mit der ID des Agenten auf die Konsole geschrieben und der Agent verabschiedet sich wieder aus dem System.

Betrachten wir nun das Buchhandel Beispiel. Dieses ist ein wenig komplexer. Es gibt zwei Typen von Agenten: den Verkäufer, der ein Buch zu einem Preis anbieten und den Käufer, der ein Buch eines bestimmten Titels zu einem möglichst günstigen Preis kaufen möchte.

```

* JADE - Java Agent DEvelopment Framework is a framework to develop

package examples.hallo;
import jade.core.Agent;

public class HalloWorldAgent extends Agent {
    // is executed on Agent startup
    protected void setup() {
        System.out.println("Hallo World! My name is "+getLocalName());

        // Make this agent terminate
        doDelete();
    }
}

```

Abbildung 3: HalloWelt Agent

Der Verkäufer hat drei behaviours: OfferRequests, PurchaseOrder und UpdateCatalogue. Ersteres feuert wenn ein Käufer eine CFP - Call for Proposal - Performative versendet. Dann wird überprüft, ob das gewünschte Buch im Angebot ist. Falls ja, wird eine Nachricht mit PROPOSE Performative und dem entsprechenden Preis an den Käufer zurückgesendet. In Java sieht dieses behaviour folgendermaßen aus:

```

private class OfferRequestsServer extends CyclicBehaviour {
    public void action() {
        MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);
        ACLMessage msg = myAgent.receive(mt);
        if (msg != null) {
            // CFP Message received. Process it
            String title = msg.getContent();
            ACLMessage reply = msg.createReply();

            Integer price = (Integer) catalogue.get(title);
            if (price != null) {
                // The requested book is available for sale. Reply with the price
                reply.setPerformative(ACLMessage.PROPOSE);
                reply.setContent(String.valueOf(price.intValue()));
            }
            else {
                // The requested book is NOT available for sale.
                reply.setPerformative(ACLMessage.REFUSE);
                reply.setContent("not-availabLe");
            }
            myAgent.send(reply);
        }
        else {
            block();
        }
    }
} // End of inner class OfferRequestsServer

```

Abbildung 4: requestOffer behaviour

```

/**
  Inner class PurchaseOrdersServer.
  This is the behaviour used by Book-seller agents to serve incoming
  offer acceptances (i.e. purchase orders) from buyer agents.
  The seller agent removes the purchased book from its catalogue
  and replies with an INFORM message to notify the buyer that the
  purchase has been sucesfully completed.
  */
private class PurchaseOrdersServer extends CyclicBehaviour {
  public void action() {
    MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.ACCEPT_PROPOSAL);
    ACLMessage msg = myAgent.receive(mt);
    if (msg != null) {
      // ACCEPT_PROPOSAL Message received. Process it
      String title = msg.getContent();
      ACLMessage reply = msg.createReply();

      Integer price = (Integer) catalogue.remove(title);
      if (price != null) {
        reply.setPerformative(ACLMessage.INFORM);
        System.out.println(title+" sold to agent "+msg.getSender().getName());
      }
      else {
        // The requested book has been sold to another buyer in the meanwhile .
        reply.setPerformative(ACLMessage.FAILURE);
        reply.setContent("not-available");
      }
      myAgent.send(reply);
    }
    else {
      block();
    }
  }
} // End of inner class OfferRequestsServer

```

Abbildung 5: purchaseOrder behaviour

Das zweite behaviour des Käufers ist die Reaktion auf einen endgültigen Kaufwunsch: `PurchaseOrder`. Dieses behaviour triggert auf Nachrichten mit `ACCEPT PROPOSAL` Performative. Sofern das Buch noch vorhanden ist und kein anderer Käufer in der Zwischenzeit gekauft hat, so wird das Buch an den Agenten verkauft und dies mit der `INFORM` Performative bestätigt. Andernfalls wird eine Nachricht mit der `REFUSE` Performative versendet, was heißt, dass die Transaktion fehlgeschlagen ist.

`UpdateCatalogue` (Abb. 6) ist ein Beispiel für ein `OneShot` behaviour, also ein behaviour, das genau einmal feuert und sich dann beendet. Es ist dafür zuständig, einen neuen Buchtitel in den Angebotskatalog aufzunehmen.

```

/**
 * This is invoked by the GUI when the user adds a new book for sale
 */
public void updateCatalogue(final String title, final int price) {
    addBehaviour(new OneShotBehaviour() {
        public void action() {
            catalogue.put(title, new Integer(price));
            System.out.println(title+" inserted into catalogue. Price = "+price);
        }
    });
}

```

Abbildung 6: Update Catalogue behaviour

Im Setup des Verkäufers (Abb. 7) wird ein Service initialisiert (book-selling) und dieser beim DFS, den gelben Seiten, registriert. Die vorher diskutierten behaviours werden eingebunden.

```

// Put agent initializations here
protected void setup() {
    // Create the catalogue
    catalogue = new Hashtable();

    // Create and show the GUI
    myGui = new BookSellerGui(this);
    myGui.show();

    // Register the book-selling service in the yellow pages
    DFAgentDescription dfd = new DFAgentDescription();
    dfd.setName(getAID());
    ServiceDescription sd = new ServiceDescription();
    sd.setType("book-selling");
    sd.setName("JADE-book-trading");
    dfd.addServices(sd);
    try {
        DFService.register(this, dfd);
    }
    catch (FIPAException fe) {
        fe.printStackTrace();
    }

    // Add the behaviour serving queries from buyer agents
    addBehaviour(new OfferRequestsServer());

    // Add the behaviour serving purchase orders from buyer agents
    addBehaviour(new PurchaseOrdersServer());
}

```

Abbildung 7: Verkäufer Setup

Der Käufer ist ein relativ simpler Agent, der einfach nur Nachrichten mit CFPs versieht und über den DFS an Agenten mit entsprechendem book-selling Service versendet. Wenn man nun JADE startet, erscheint zunächst die leere JADE GUI ohne Agenten, aber mit DFS und AMS:

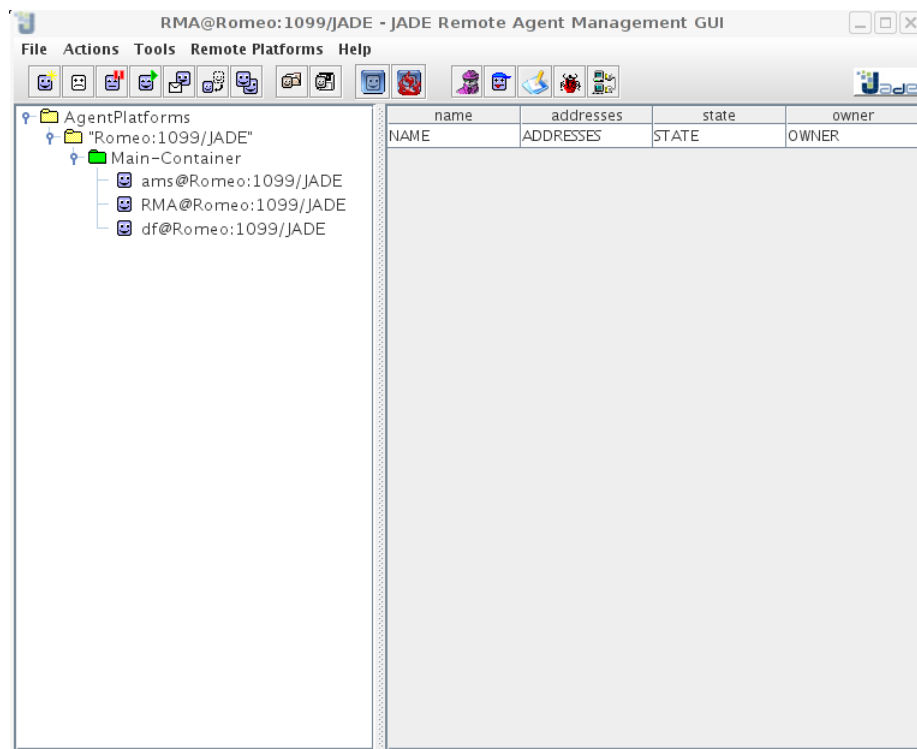


Abbildung 8: JADE GUI

Lässt man nun das Buchhandel Beispiel ablaufen, so sieht die Kommunikation mit Verkäufer Jon und Käufer Carl wie in Abb. 9 aus.

Dieses anschauliche Beispiel zeigt, dass es mit relativ wenig Aufwand möglich ist, mit Hilfe von JADE ein Agentensystem zu planen und zu implementieren. Auch die Ausführung des Systems ist über die anwenderfreundliche GUI kein Problem.

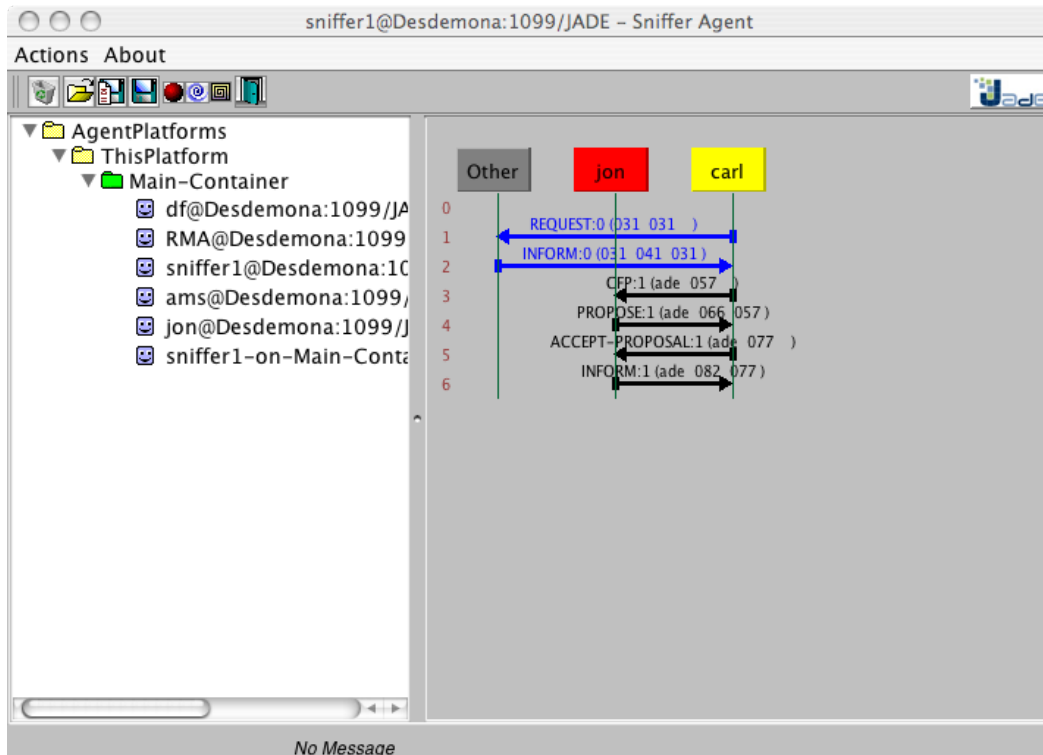


Abbildung 9: Buchhandel Verlauf

## 4 Zusammenfassung und Diskussion

Diese Arbeit präsentierte die Java Middleware JADE, welche eine komfortable Möglichkeit ist, FIPA-standardisierte und plattformunabhängige Multiagentensysteme zu implementieren und auszuführen. Der FIPA Standard, welcher eine Norm für Agentenarchitektur und -kommunikation ist, wurde kurz erläutert. Anhand einfacher Beispiel wurde die Java Implementation eines Multiagentensystems mit JADE demonstriert und erklärt. JADE bietet eine fertige GUI, welche die Agentenplattform beinhaltet. Aufgabe eines Programmierers ist es noch, die jeweiligen Agenten zu implementieren, mit behaviours und Diensten zu versehen und einzubinden. Die Kommunikation befolgt strikt FIPA und benutzt somit die ACL. Ergo stellt es auch kein Problem dar, eine JADE Plattform in ein fremdes System zu integrieren, sofern auch diese nach FIPA konstruiert wurde.

Zusammenfassend würde ich JADE als gutes und komfortables Tool beurteilen.

Die Anwendung ist meines Erachtens einfach und für einen einigermaßen erfahrenen Java Programmierer kein Problem. Die GUI ist übersichtlich und selbsterklärend. Eine nette Funktion ist der integrierte Sniffer, mit dem man die Kommunikation der Agenten verfolgen kann. Ein Artefakt weist JADE mit Standardeinstellungen allerdings auf. Es können schonmal einige Minuten vergehen, bis eine Agentenkommunikation zustande kommt. Sollte dies generell der Fall sein und nicht durch Parameter regulierbar, könnte dies zu Performanzeinbußen führen. In einem kleinen System ist dies kein Problem, wohingegen das in größeren Systemen durchaus problematisch sein kann. Somit sollte man also im Vorfeld kritisch sein, ob diese Verzögerung steuerbar ist.

## Literatur

- [1] F. Bellifemine, G. Caire, A. Poggia, and G. Rimassa. *Jade - A White Paper*. <http://jade.tilab.com>.
- [2] G. Caire. *Jade Tutorial - Jade Programming for Beginners*. <http://jade.tilab.com>.
- [3] FIPA-Organization. *FIPA Specification*. <http://www.fipa.org>.
- [4] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.
- [5] Wiki-Community. *Wikipedia*. <http://wikipedia.org>.