Universität Münster
Institut für Informatik

Verteilte Programmierung eines Agentensystems mit LIME

Evaluationsbericht

vorgelegt von

Simon Prinzleve, Christoph Tepper, Steffen Wachenfeld

im

September 2002

Seminarleitung:

Dr. Dietmar Lammers,
Prof. Dr. Guido Wirtz

Inhaltsverzeichnis

1 Einleitung

Agentenorientierte Softwareentwicklung ist eine der neuesten Entwicklungen in der Softwareentwicklung und Gegenstand zahlreicher Forschungsprojekte. Der Wunsch, die agentenorientierte Softwareentwicklung zu einem neuem Paradigma für die Softwareindustrie zu machen, legen die Entwicklung von Frameworks nahe, welche die Erstellung und den Betrieb von (verteilten) Agentensystemen erleichtern.

Ziel derartiger Frameworks ist es, die Wiederverwendbarkeit von Implementierungslösungen, insbesondere solcher der unteren Ebene zu erhöhen. Im speziellen sollte ein Framework zur Erstellung von verteilten Agentensystemen die Implementierung der Hauptaspekte von Agenten unterstützen. Die Hauptaspekte sind die Realisierung von autonomem, proaktivem Verhalten, sowie die Unterstützung von Kommunikation zwischen Agenten. Darüberhinaus ist die Bereitstellung einer Laufzeitumgebung für Agenten durch ein solches Framework wünschenswert. In diesem Kontext, sind z.B. die Unterstützung von Mobilität oder Hilfsmittel zum Debugging sinnvolle Eigenschaften.

Die Zahl der hierzu existierenden Frameworks ist entsprechend der dem agentenorientierten Softwareentwicklung gewidmeten Aufmerksamkeit gross. Daher wurden im Rahmen eines Seminars im Fachbereich Informatik an der Westfälischen Wilhelmsuniversität Münster zahlreiche Frameworks[1] einer Bewertung hinsichtlich der oben genannten Kriterien unterzogen. Aus diesen wurden in einer Vorauswahl wenige, vielversprechende Frameworks ausgewählt. Diese Frameworks wurden ausführlicher untersucht und es wurden Testimplementierungen von Agentensystemen mit ihnen durchgeführt. Als eines der vielversprechenden Frameworks wurde LIME ausgewählt.

Die vorliegende Arbeit beschreibt die im Rahmen dieses Seminars vorgenommene Bewertung von LIME als Framework zur Realisierung von Multiagenten Systemen. Ziel dieser Arbeit ist es, eine detaillierte und praktisch fundierte Evalutation zu erarbeiten. Hierzu wurde ein agentenbasiertes Online-Rollenspiel mit LIME realisiert.

Wir werden zunächst LIME allgemein vorstellen und eine Einführung in die dabei verwendenten Konzepte geben sowie erste Bewertungen zu ausgewählten Teilen vornehmen. Anschließend folgt eine Beschreibung des realisierten Spiels und der Implementierung des Spiels mit Hilfe von LIME. Die dabei gemachten Erfahrungen werden anschließend anhand der zentralen LIME-Konzepte vorgestellt und zur Bewertung des Frameworks herangezogen.

[1] Auflistung unter www.agentlink.org\resources\agent-software.html

2 Lime

LIME - *Linda in a mobile environment* ist eine Erweiterung des *Linda*-Konzeptes. In diesem Abschnitt werden zunächst die Grundgedanken von *Linda* aufgeführt. Es folgt eine Beschreibung der Konzepte, um die LIME *Linda* erweitert. Anschließend wird LIME – soweit dies unabhängig von konkreten Projekterfahrungen möglich ist – bewertet. Diese Bewertung ist in eine konzeptionelle und eine technische Bewertung unterteilt.

2.1 Linda

Linda ist ein Koordinationskonzept für nebenläufige bzw. verteilte Prozesse, das auf dem Prinzip der gemeinsamen Speichernutzung beruht. Die Motivation besteht darin, orthogonale Aspekte der Programmierung – verteilungsspezifische und applikationsspezifische – zu trennen, und dadurch die Wiederverwendbarkeit von Koordinationslösungen zu erhöhen.

Die zentrale Komponente des Linda-Interaktionsmodells ist der gemeinsame Speicher, welcher in Form eines Tupelraums realisiert ist. Ein Tupelraum ist eine Multimenge von Tupeln. Ein Tupel ist eine Sequenz getypter Einträge, z.B. ("Agentenspiele"; "TinyD"; 2002). Diese Einträge werden als *actuals* bezeichnet.

Auf dem Tupelraum können drei Arten von Operationen ausgeführt werden: Es ist möglich ein Tupel einzufügen, zu entnnehmen oder zu lesen.
Das Einfügen eines Tupels t geschieht durch die *out(t)* Operation. Das Entnehmen und das Lesen von Tupeln wird durch die *in(p)* bzw. die *rd(p)* Operation getätigt. Der Parameter p wird als *Template* bezeichnet. Ein Template unterscheidet sich von einem Tupel dadurch, dass es neben actuals auch *formals* enthält.[2] Formals sind getypte "Wildcards" – sie dienen als Platzhalter für beliebige Actuals des entsprechenden Typs. Das Ergebnis von *in(p)* bzw. *rd(p)* ist ein Tupel t, welches nichtdeterministisch aus der Multimenge der zum Template p passenden Tupel ausgewählt wird. Ein Tupel t passt zu einem Template p, falls beide die gleiche Länge aufweisen, und die einzelen Einträge zueinander passen. Ein Eintrag von t passt zu einem Eintrag von p wenn beide das gleiche Actual aufweisen, oder das Actual von t vom Typ des Formals an der enstprechenden Stelle in p ist. Das Template ("Agentenspiele"; ?String; ?Integer) passt z.B. zu obigem Beispieltupel
Sowohl *in(p)* als auch *rd(p)* sind blockierende Operationen. Das bedeutet,

[2] Ein Template genügt ebenfalls der mathematischen Definition des Tupels. Wenn im Folgenden von Tupeln gesprochen wird, ist jedoch jeweils immer ein nur aus Actuals bestehendes (Daten-)Tupel gemeint

dass der ausführende Prozess solange blockiert, bis ein zu p passendes Tupel gefunden wird.

Zu den oben angegebenen Basisoperationen sind einige Erweiterungen denkbar. Dies sind z.B. nichtblockierende Varianten von *in(p)* und *rd(p)*, welche oft als *inp(p)* und *rdp(p)* bezeichnet werden. Darüberhinaus sind Operationen *ing(p)* bzw. *rdg(p)* denkbar, welche jeweils alle zu p passenden Tupel als Ergebnis liefern. Eine zusätzliche denkbare Erweiterung stellen Operationen mit Zeitstempeln dar.

2.2 Das Lime Modell

Agenten und Mobilität Die in Kap. 2 vorgestellten Konzepte behandeln die Koordination der Interaktionen von Kommunikationspartnern. Die Kommunikationspartner sind im hier betrachteten Rahmen mobile Agenten. Im Folgenden werden kurz die Aspekte der *Mobiliät* eines mobilen Agentens diskutiert.

Die Mobilität eines Agenten kann logischer oder physischer Natur sein. *Logische Mobilität*[3] bezeichnet den serialisierten Transfer eines aktiven Agenten-Objekts in Form seines Quellcodes sowie seiner zustandsbestimmenden Daten. Darüber hinaus ist die Re-Aktivierung des Objekts auf der empfangenden Seite – üblicherweise in Form eines Threads – kennzeichnend für logische Mobilität. Der Agent ist also in dem Sinne mobil, als dass er – als eine zustandsbehaftete und aktive Einheit – den Host wechseln kann.

Der Ausdruck *Physische Mobilität* bezieht sich auf die Mobilität des mobilen Hosts auf dem sich der Agent befindet. In diesem Fall ist der Agent passiv mobil. Der Host[4] des Agenten bewegt sich in der physischen, realen Welt. Diese Bewegung wirkt sich dabei auf den Verbindungszustand des auf dem Host laufenden Agenten aus. Durch die Änderng des Verbindungszustandes erfährt der Agent also indirekt die Folgen der physischen Bewegung des Hosts. Insbesondere hat ein Verbindungsabbruch zu seiner bisherigen Kommunikationsgemeinschaft zur Folge, dass er sich eine neue Kommunikationsgemeinschaft suchen muss.

Vor allem die physische Mobilität wirkt sich also auf den Zustand von Kommunikationsgemeinschaften aus. Sie hat aus Sicht der Gemeinschaft zur Folge, dass zu beliebigen Zeitpunkten *unbekannte* neue Mitglieder hinzukommen können, sowie dass Mitglieder die Gemeinschaft jederzeit verlassen können.

[3] Der Begriff "logische Mobilität" wird hier synonym zum später verwandten Begriff "physische Migration" gebraucht.

[4] z.B. ein Handy, ein PDA oder ein Laptop

Die Zusammensetzung von Kommunikationsgemeinschaften ist im mobilen Umfeld also höchst dynamisch.

Adaption von *Linda* an einen mobilen Kontext Wie in Kap. 2.1 gezeigt wurde, ermöglicht das Linda-Konzept sowohl zeitlich als auch räumlich asynchrone Kommunikation. Darüber hinaus brauchen Sender und Empfänger einander nicht zu kennen; Kommunikation ist in *Linda* also anonym möglich.
Diese Eigenschaften lassen das Linda-Konzept gerade für eine hoch dynamische Umgebung effektiv erscheinen. Wie oben aufgeführt wurde, ist gerade der mobile Anwendungskontext eine Umgebung mit eben dieser Eigenschaft. Deshalb entstand im Rahmen eines Forschungsprojekts LIME, eine Adaption bzw. Erweiterung des Linda-Konzepts für mobile Umgebungen.

Wie unter "Agenten und Mobilität" gezeigt, hat Mobilität eine sich dynamisch verändernde Kommunikationsgemeinschaft zur Folge. Der Inhalt (bzw. die Daten) einer solchen Gemeinschaft setzt sich aus den Daten der einzelnen Mitglieder dieser Gemeinschaft zusammen. Dementsprechend hat die Dynamik der Zusammensetzung von mobilen Gemeinschaften unmittelbar einen sich dynamische verändernden Inhalt der Gemeinschaft zur Folge. Um diesem Aspekt Rechnung zu tragen, besteht die Grundidee von LIME darin, den Tupelraum in atomare Tupelräume aufzubrechen, wobei jeder dieser atomaren Tupelräume fest mit einem mobilen Agenten assoziiert wird. "Fest assoziiert" bedeutet, dass ein mobiler Agent seinen Tupelraum im Falle der Mobilität mit sich nimmt. Für diese elementaren Teiltupelräume bestehen Regeln für das vorübergehende bzw. *transiente* Verbinden dieser Tupelräume, welche auf dem Verbindungszustand einer Gemeinschaft beruhen.

Ein Agent verfügt über mindestens einen atomaren Tupelraum. Ein derartiger atomarer Tupelraum wird als *Private Tuple Space* bezeichnet. Jeder atomare Tupelraum ist benannt. Jeder Agent hat nun die Möglichkeit seinen[5] *Private Tuple Space* zu veröffentlichen[6]. Aus veröffentlichten Tupelräumen ergeben sich Tupelräume höherer Ebenen, indem jeweils veröffentlichte Tupelräume gleichen Namens verbunden werden. Tupelräume deren besitzende Agenten sich auf einem Host befinden werden zunächst zu dem als *Host Level Tuple Space* bezeichneten Tupelraum verbunden. Sind mehrere Hosts miteinander verbunden, so werden die ihnen zuordbaren Host

[5] im Folgenden wird zur Vereinfachung der Singular verwendet. Prinzipiell kann ein Agent jedoch über mehrere Tupelräume verfügen

[6] Die Autoren von LIME sprechen hier von *sharing*.

Level Tuple Spaces transitiv zu einem *Federated Tuple Space* verbunden.

Der Zugriff eines Agenten auf einen Tuplespace erfolgt nun mittels der gewöhnlichen Linda Operationen auf seinen atomaren Tupelraum. Ist dieser Tupelraum veröffentlicht, hat dies je nach Verbindungsstatus, einen Zugriff auf den Host Level - bzw. Federated Tupel Space des entsprechenden Namens zur Folge. Die Reichweite seiner Operationen, d.h. die Anzahl der beteiligten Hosts bzw. Agenten, ist dabei für den ausführenden Agenten transparent.

Das Prinzip der verübergehenden Verbindung (auch: *transient sharing*) von atomaren Tupelräumen hat zur Folge, dass sich der Inhalt eines Tupelraumes aus Sicht eines Agenten dynamisch ändert.

Explizite Kontextkontrolle Die eindeutige Identifizierung eines Agenten, welche durch die Netzwerkadresse seines Hosts zuzüglich einer hostweit eindeutigen ID zusammensetzt, wird als *location* bezeichnet. Führt ein Agent eine lesende Operation auf "dem" Tupelraum aus, werden die Tupel aller verbundenen Agenten – unabhängig von deren Location – geprüft und in Betracht gezogen. Prinzipiell bietet LIME also völlige Abstraktion von der Location von Agenten, bzw der Location von Tupeln. Darüber hinaus bietet LIME auch Varianten der Linda Basisoperationen (out, rd und in) mit zusätzlicher expliziter Angabe von Locations. Dies dient laut den Autoren von LIME lediglich der Optimierung der Performanz. Es sind jedoch auch Anwendungen denkbar, die eine Angabe der Location aufgrund des LIME-Modells zwingend erfordern.[7]

Um ein Tupel in einen bestimmten atomaren Teiltupelraum eines Tupelraumes höherer Ordnung zu schreiben, existiert die *out[λ](t)* Operation, welche die Basisvariante um den Parameter λ erweitert. Dieser Parameter λ bezeichnet die Ziel-Location des ausgegebenen Tupels. Diese erweiterte Operation wird in zwei Schritten vollzogen. Im ersten Schritt wird das Tupel t – wie bei der herkömmlichen out-Operation – im Private Tuple Space des ausgebenden Agenten ω positioniert. Nun verfügt das Tupel über zwei Location Informationen: Die *current location* ω und die *destination location* λ. Sobald ω mit λ verbunden wird, wird das Tupel im zweiten Schritt von ω zu λ bewegt. Besteht diese Verbindung bereits zum Zeitpunkt der Ausführung von *out[λ](t)*, erfolgt die Durchführung der Schritte eins und zwei atomar.

Von den Operationen *in* und *out* existieren ebenfalls Varianten mit expliziter Angabe von Locations. Im Speziellen lauten diese Operationen *in[λ, ω](t)* und *rd[λ, ω](t)*, wobei λ und ω wieder die Destination Location und die Current Location angeben. Hier besteht jeweils die Möglichkeit

[7] vgl. "logische Migration"in TinyD

einen Parameter unspezifiziert[8] zu lassen.

Zusätzlich bietet LIME nicht-blockierende Varianten der lesenden Operationen, *inp[λ, ω](t)* und *rdp[λ, ω](t)*, sowie Operationen, die alle passenden Tupel als Ergebnis liefern, *ing[λ, ω](t)* und *rdg[λ, ω](t)*. Bei diesen Operationen ist die Angabe der Current Location obligatorisch. Dies ist dadurch begründet, dass die semantisch korrekte Ausführung dieser Operationen auf den gesamten Tupelraum jeweils eine Transaktionssperre über die gesamte Kommunikationsgemeinschaft erfordern würde. Es sei an dieser Stelle angemerkt, dass im Rahmen der Angabe einer Current Location auch die Angabe der Location eines Hosts (*host location*) anstelle der Location eines Agenten möglich ist.

Reaktive Programmierung mit Lime Wie unter "Agenten und Mobilität" gezeigt wurde, hat Mobilität eine stark dynamische Umwelt zur Folge. In einer solchen Umwelt stellen Reaktionen auf Veränderungen einen wesentlichen Aspekt von Applikationen dar. Daher erweitert LIME das Basiskonzept von Linda um das Konzept der *reaction*. Eine Reaction $\mathcal{R}(s, p)$ wird durch ein Codefragment s bestimmt, das ausgeführt wird, wenn das Template p in dem der Reaction zugeordneten Tupelraum gefunden wird. Jedes Mal, wenn eine Operation auf einem Tupelraum ausgeführt wird[9], werden alle diesem Tupelraum zugeordnetem Reactions in zufälliger Reihenfolge betrachtet. Dabei wird jeweils das Template p der aktuellen Reaction im Tupelraum gesucht. Falls p gefunden wird, führt dies zur Ausführung von s. Dabei ist zu beachten, dass in s – um einen Deadlock zu vermeiden – keine blockierenden Operationen erlaubt sind.

Soll gewährleistet sein, dass das Auffinden von p und die Ausführung von s atomar durchgeführt werden, ist hierzu eine Transaktion notwendig. Bezieht sich eine Reaction auf einen Federated Tuple Space, erfordert dies eine verteile Transaktion über die gesamte den Tupelraum bildende Gemeinschaft. Da dies eine sehr unperformante und damit unpraktikable Lösung darstellt, bietet LIME zwei Arten von Reactions. Dies sind die *weak Reaction* und die *strong reaction*. Letztere bietet die angesprochene Atomarität. Allerdings erfordert sie die explizite Angabe von Locations, und kann sich damit maximal auf einen Host-Level Tuple Space beziehen.

[8] konkret: es wird eine Konstante als Parameter übergeben, welche als Platzhalter für jede beliebige Location dient

[9] Prinzipiell sollte es eigentlich genügen, lediglich schreibende Operationen zu betrachten.

2.3 Bewertung

Die bis hierhin getätigten Ausführungen erlauben bereits eine erste Bewertung von LIME. Vor allem die konzeptionellen Aspekte von LIME bzw. *Linda* können ohne konkreten Anwendungskontext evaluiert werden. Dies soll im Folgenden ausgeführt werden. Daran anschließend folgen einige wertende Anmerkungen zu Implementierungsdetails von LIME, welche ebenfalls anwendungsunabhängig gültig sind.

2.3.1 Konzeptionelle Bewertung

Die konzeptionelle Bewertung erfolgt in zwei Teilen. Zunächst wird das "shared Memory" Konzept *Linda* bewertet. Anschließend wird kurz auf die von LIME gemachten Erweiterungen eingegangen.

Linda Das Linda-Konzept weist eine Reihe von Vorteilen auf. Aufgrund des gewährleisteten Abstraktionsgrades ist die Interaktion unabhängig von der Kenntnis über die Interaktionspartner (anonyme Kommunikation), von der Kenntnis über den Zeitpunkt der Interaktion (asynchrone Kommunikation) sowie von der Kenntnis über die räumliche Herkunft der Information. Ein weiterer wichtiger Punkt ist die Einfachheit von *Linda*. Die Reduktion auf wenige, einfache Operationen erleichtert das Vermeiden von Fehlern durch erhöhte Transparenz und ermöglicht eine rasche Entwicklung von Applikationen. Weiterhin weist das Tupel Konzept eine hohe Konsistenz zu dem Paradigma der Objektorientierung auf. Außerdem ist das Linda-Konzept derart mächtig, dass sich andere Interaktionsparadigmen wie z.B. "Message Passing", "globale Variablen", "Gruppenkommunikation" und "Client/Server-Kommunikation mit Remote Procedure Calls" damit simulieren lassen.

Das Linda-Konzept eines gemeinsamen Speichers weist folgende Nachteile auf: Ohne zusätzliche Erweiterungen, bestehen deutliche Sicherheitsprobleme, da alle beteiligten Kommunikationspartner beliebig den Datenraum manipulieren können. Außerdem erfordert die asynchrone Kommunikation in der Grundversion von *Linda* (ohne Zeitstempel) ggfs. die Bereinigung des Tupelraums um veraltete Nachrichtentupel, also ein explizites *garbage collection*. Weiterhin erfordern einige Anwendungsfälle andere Formen der Kommunikation, z.B. synchrone Kommunikation. Die Simulation anderer Interaktionsparadigmen (s.o.) ist mit *Linda* zwar prinzipiell möglich, jedoch jeweils aufwändiger als die jeweils originäre Variante.

Lime Die wesentliche Erweiterung *Linda*s durch LIME ist das Konzept der Reactions. Aufgrund der hohen Dynamik in mobilen Umgebungen ist dieses Konzept als höchst sinnvoll einzustufen. Mögliche Kritikpunkte an diesem Konzept ergeben sich weniger durch das Konzept selbst, als vielmehr durch die Praxisnähe der Umsetzung. Vergleiche hierzu Kapitel 3.4.3.

2.3.2 Technische Bewertung

Die folgende technische Bewertung von LIME konzentriert sich vorwiegend auf Implementierungsdetails[10], die als problematisch empfunden werden.

Die wesentliche potentielle Problemquelle LIMEs ist in der Gestaltung des *engagement* Prozesses zu sehen. Dieser weist folgende Schwächen auf:

- LIME ermöglicht nur das Engagement jeweils eines LIME-Servers mit einer Gemeinschaft. Dies bedeutet, dass eine Gemeinschaft nur "Individuen" aufnehmen kann, keine bereits bestehenden Gruppen.

- Dabei ist es so, dass die einzelnen Individuen jeweils nur sequentiell aufgenommen werden können. Versuchen mehrere Individuen gleichzeitig ein Engagement, führt dies zum Absturz.

- Eine LIME-Gemeinschaft erfordert die Deklaration eines *leaders*. Diese Deklaration erfolgt nicht dynamisch, sondern muss jeweils explizit vom Anwender vorgenommen werden.

- Zur Kommunikation während des Engagements wird das *Broadcast*-Verfahren benutzt. Dieses erfordert das Senden von Daten an eine bestimmte in LAN-Netzten für Broadcast reservierte Adresse. Dieses Verfahren ist jedoch nicht in größeren Netzen möglich, insbesondere nicht im Internet.

LIME ermöglicht es, komplette serialisierbare Objekte im Tupelraum als Eintrage eines Tupels abzulegen. Das "matching" bedient sich dabei des bekannten Ansatzes des Überschreibens der *equals*-Methode der Klasse "Object". Die zu kommunizierenden Objekte werden dabei sowohl beim Schreiben in den Tupelraum, als auch beim Lesen geklont. Dabei wird von LIME jedoch das Vererbungskonzept des Objektorientierten Paradigmas nicht unterstützt, Informationen über die Vererbungshierarchie werden nicht verwendet. Es ist daher nicht möglich per entsprechendem Template Superklassen mit Subklassen zu matchen. Aus selbigem Grund ist es nicht möglich,

[10] Nochmaliger Hinweis: dieser Abschnitt widmet sich der Implementierung *von* LIME - nicht einer Implementierung *mit* LIME. Zu letzterem Vergleiche die folgenden Kapitel.

die Klasse “lights.adapters.Tuple” zu erweitern und sich somit eine eigene Hierarchie von Tupelklassen zu erstellen.

Ein weitere Schwachstelle LIMEs ist die Unterstützung der logischen Mobilität bzw. der physischen Migration. In der Dokumentation verweisen die Autoren von LIME auf die Benutzung der “μCode”-Bibliothek[11]. Die Integration selbiger gestaltet sich jedoch als schwierig. Zu detaillierten Beschreibungen der Probleme bei der Anwendung vgl. Kap. 3.4.5.

Um sicherzustellen, dass ein Tupelraum jeweils nur von einem Agenten besessen wird, gestattet LIME jeweils nur dem Thread des besitzenden Agenten, auf einen Tupelraum zuzugreifen. Dieses aus theoretischer Sicht sinnvoll wirkende Konzept führt bei praktischen Anwendungen zu Problemen. Dies gilt besonders für Anwendungen mit wirklich autonom handelnden Agenten. Nähere Erläuterungen hierzu finden sich in Kap. 3.4.2

Positiv anzumerken ist, dass LIME architektonisch durch Verwendung einer Adapter-Schicht recht flexibel gehalten wurde. So ist es theoretisch möglich, z.B. die Klassen für den Tupelraum oder für die Tupel auszutauschen, soweit hierfür “Adapterklassen” vorliegen.

3 tinyD

tinyD ist ein agentenbasiertes Rollenspiel, welches mit LIME implementiert wurde. Die Implementierung eines Spiels mit LIME hat zum Ziel, die theoretische Bewertung des Frameworks um Erfahrungen aus dem praktischen Umgang zu bereichern. Mit tinyD is ein größeres Spiel entstanden, welches die Verwendung einer Verhaltenssteuerung von AI-gesteuerten Agenten sowie einer recht mächtigen Kommunikation vorraussetzt. Die Agenten können zudem eine große Menge von Zuständen einnehmen, die sich direkt auf die Aktionsmöglichkeiten auswirkt. Der Mächtigkeit des Spiels ist es zu verdanken, das sich einige Schwachstellen und Stärken von LIME überhaupt haben herausstellen können.
In diesem Kapitel wird das Konzept (Abschnitt 3.1) und die Architektur (Abschnitt 3.2) von tinyD aus softwareentwicklungstechnischer Sicht beschrieben sowie die bei der Implementierung gemachten Erfahrungen dargestellt.

[11] muCode.sourceforge.net

3.1 Spielkonzept

Ein Spiel, dessen Implementierung bewertende Aussagen über ein verwendetes Framework wie LIME zulässt, erfordert einen gewissen Umfang und ein Konzept, welches möglichst viele Ansprüche an das verwendete Framework stellt.
Aus diesem Grund haben wir uns mit tinyD für die Entwicklung eines agentenbasierten Multi-Player-Rollenspiels entschieden. Ziel war es nicht, ein marktreifes Rollenspiel zu erstellen, sondern ein Spiel, welches den oben angesprochenen Umfang hat und an LIME gewisse Anforderungen stellt.
In tinyD laufen mehrere Charaktere durch Dungeons (Labyrinte), begegnen sich dort und können interagieren. Die Charaktere können sowohl von einem menschlichen Spieler, als auch von einer AI gesteuert werden können.
Die Steuerung lässt sich jederzeit ändern, ein menschlicher Spieler kann die Steuerung eines AI-gesteuerten Charakters übernehmen und umgekehrt kann die Steuerung von einem Menschen an eine AI abgetreten werden. Ein menschlicher Spieler kann dabei beliebig viele Charaktere steuern und beliebig viele AI-gesteuerte Charaktere beobachten.
Die Anzahl der Dungeons ist flexibel (auch zur Laufzeit) und die Dungeons dürfen sich auf beliebigen Rechnern befinden. Gibt es mehr als einen Dungeon, dann ist es möglich mit den Charakteren zwischen den Dungeons hin und her zu wechseln.
Ein Dungeon besteht aus beliebig vielen dreidimensional angeordneten Räumen. Räume können miteinander verbunden sein, z.B. durch Türen oder Treppen zwischen benachbarten Räumen, magischen Portalen zwischen beliebigen Räumen eines Dungeons oder durch "Warp Gates", die die Verbindungen zwischen Räumen zwei verschiedener Labyrinte herstellen.
Charaktere haben bestimmte Eigenschaften. Jeder Charakter hat einen bestimmten Typ (z.B. BladeMaster oder DemonHunter), sowie einen gewissen Level, der den Erfahrungsstand repräsentiert. Die Unversehrtheit eines Charakters wird durch eine Anzahl von verbleibenden Lebenspunkten ausgedrückt und der Schaden, den ein Charakter macht durch einen Schadenswert.
Zwischen Charakteren, die sich im selben Raum befinden existieren verschiedene Interaktionsmöglichkeiten. Ein Charakter kann einen anderen attackieren bzw. heilen, wobei Schadenswert mit Lebenspunkten verrechnet werden. Ein Charakter kann in einem Raum reden, so dass alle Charaktere die sich in ihm befinden dies hören. Soll ein Charakter mit einem anderen Charakter unter vier Augen reden, so kann er hierzu einen Dialog mit dem anderen führen (dies ist auch mit AI gesteuerten Charakteren möglich, siehe 3.3.1).

Unser Spielkonzept hat weiter vorgesehen, Gegenstände finden und verwenden zu können, sowie Kleidung, Rüstung usw. Da die Implementierung dieser Features zwar nett, aber im Hinblick auf die Verwendung und Bewertung von LIME keine weiteren Erkenntnisse gebracht hätte, sind diese zwar in die Architektur mit eingeflossen, wurden aber nicht komplett ausimplementiert.

3.2 Architektur von tinyD

In diesem Abschnitt wird die Architektur von *tinyD* beschrieben. Die für *tinyD* gewählte Architektur wurde auf die Verwendung von LIME ausgerichtet. Das Ergebnis ist eine Architektur aus zwei Schichten, die durch Interfaces gekoppelt sind.
Auf der Interaktionsschicht von *tinyD* interagieren Agenten mit Hilfe von LIME nach dem *Linda*-Konzept. Diese Agenten sowie ihre Fähigkeiten und Aufgaben in dem System werden im folgenden genauer beschrieben.
Über der Interaktionsschicht liegt die Steuerungsschicht. In dieser Schicht liegen Komponenten, die die darunterliegenden Agenten steuern oder beobachten. Die Komponenten dieser Schicht werden im folgenden ebenfalls genauer beschrieben.
Die Verbindung der beiden Schichten wurde durch Interfaces realisiert, so dass die Komponenten der beiden Schichten getrennt entwickelbar und austauschbar sind.

3.2.1 Interaktionsschicht

Die Interaktionsschicht ist die für die Betrachtung von LIME interessanteste Schicht. Sie enthält die Agenten, die nach dem *Linda*-Konzept interagieren. Dies sind der DungeonMaster-Agent sowie der Charakter-Agent.

DungonMaster Der Agent DungeonMaster, implementiert durch die Klasse `AC_DungeonMaster`, stellt ein Spielumfeld (einen Dungeon) und Spieler-Rollen zur Verfügung. Mit jedem DungeonMaster, der gestartet wird, entsteht ein Dungeon. Ein solcher Dungeon ist von Charakter-Agenten betretbar. Existieren mehrere solcher Dungeons, z.B. durch starten mehrerer DungeonMaster in einem Netzwerk/auf einem Rechner, so werden die Dungeons an bezeichneten Verknüpfungspunkten durch "Warp-Gates" verbunden.

Zum DungeonMaster gehören zwei Tuplespaces. Der `GlobalWorldTS` ist ein Tuplespace, den alle DungeonMaster sharen und unter gleichem Namen

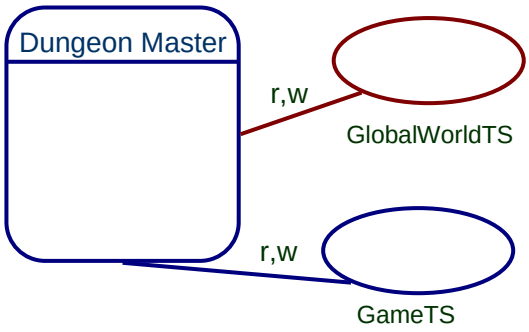

Abbildung 1: Architektur des DungeonMaster Agenten

zur Verfügung stellen. In diesem Tupelspace finden beispielsweise Charakter-Agenten, die einen Dungeon betreten wollen, DungeonInformation-Tupel mit allen notwendingen Informationen. Jeder DungeonMaster legt in seine Partition des `GlobalWorldTS` ein solches Tupel mit den entsprechenden Informationen für das Betreten seines Dungeons.
Der andere Tuplespace des DungeonMasters ist der `GameTS`, er enthält die konkreten Daten des Dungeons. Dieser Tupelspace wird nur mit Agenten geshared, die den Dungeon betreten haben. In ihm befinden sich Informationen über die Räume und die Verbindungen. Ausserdem wird dieser Tupelspace vom DungeonMaster für die Kommunikation mit den Spielern genutzt. Bei der Benennung dieses Tuplespaces wurde durch die Einbeziehung der AgentID des DungeonMaster-Agenten in den Namen des Tuplespaces sichergestellt, dass im gesamten Spielsystem nur ein Tuplespace dieses Names existiert.

Character Der Charakter-Agent, implementiert durch die Klasse `AC_Character` repräsentiert eine Spielfigur. Er kann Dungeons betreten und shared dann mit dem entsprechenden DungeonMaster-Agenten und allen anderen Charakter-Agenten in diesem Dungeon einen `GameTS`-TupleSpace. Sämtliche Interaktion mit Agenten dieses Dungeons findet über diesen TupleSpace statt.

Die Klasse `AC_Character` implementiert die Interfaces `I_ACObservable` sowie `I_ACCommandable`. Diese Interfaces garantieren das Vorhandensein von Konstanten und Methoden, die zur Steuerung (und Beobachtung) des Agen-

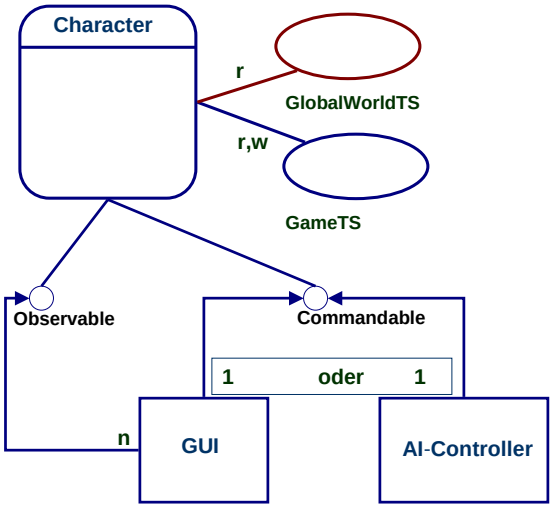

Abbildung 2: Architektur des Charakter Agenten

ten notwendig sind.
An jedem Charakter-Agenten kann sich ein Commander anmelden um diesen zu steuern sowie beliebig viele Observer, die den Charakter beobachten können.

3.2.2 Steuerungsschicht

Die GUI-Komponenten zur Steuerung von (einem oder mehreren) Charakter-Agenten durch den Benutzer, sowie die AI-Komponenten zur Steuerung durch den Computer gehören in die Steurungsschicht. Diese Komponenten implementieren die Interfaces `I_ACCommander` und `I_ACObserver`, die es ihnen erlauben, sich bei den Agenten als *Commander* bzw. *Observer* anzumelden.
Die Beobachtung und Steuerung der Charakter-Agenten findet ausschließlich durch die Nutzung von Interfaces statt und ist somit komplett von LIME getrennt. Die zur Verfügung stehenden Methoden sind für alle Steuerungskomponenten gleich, somit hat eine AI-Steuerungskomponente exakt die gleichen Möglichkeiten wie eine Benutzer-Steuerungskomponente. Ebenso kann ein Charakter-Agent nicht unterscheiden, ob er von einer AI oder von einem Benutzer gesteuert wird. Dies ist absichtlich so und hat sich be-

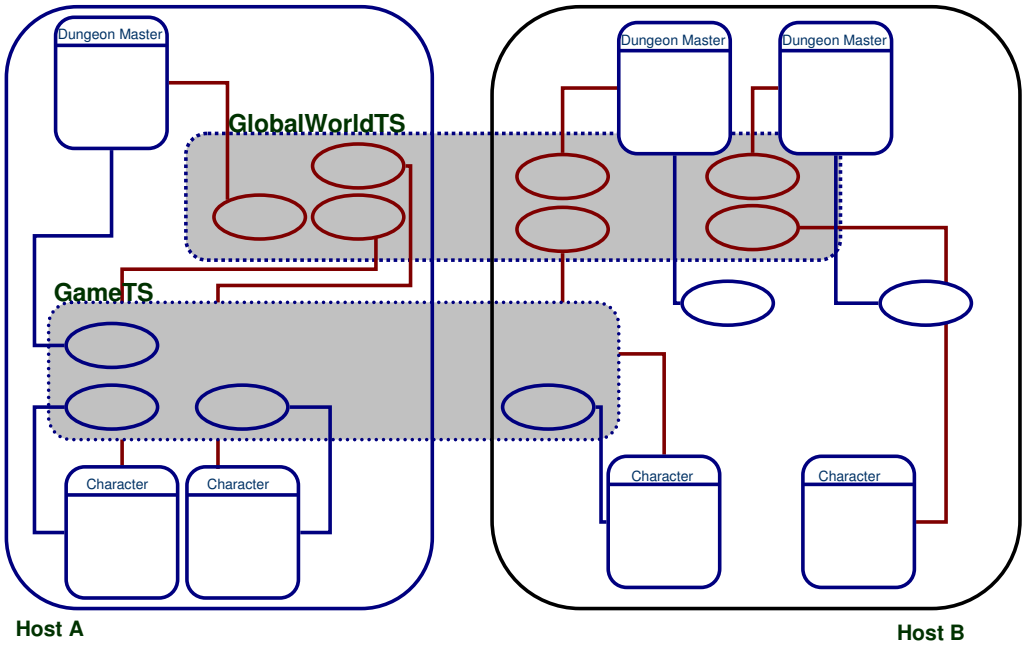

Abbildung 3: Zusammenspiel von Charakter und DungeonMaster Agenten

sonders hinsichtlich der Übertragung von Steuerung von AI an Benutzer und umgekehrt sehr bewährt. Die Steuerungskomponenten sind dadurch getrennt weiterentwickelbar und bei Bedarf einfach austauschbar.

3.3 Steuerung der Charakter-Agenten

Zum Erhalt eines vollwertigen Agentensystems gehört neben der Steuerung von Agenten durch einen Benutzer die Steuerung durch eine AI. Die AI-Komponente von tinyD stattet die Agenten mit reaktivem und proaktivem Verhalten aus, wodurch die Agenten zum autonomen Handeln befähigt werden. Die Realisierung der Verhaltensimplementierung sowie die Steuerung durch ein GUI werden im folgenden beschrieben.

3.3.1 AI-Steuerung

In diesem Abschnitt wird dargestellt, wie LIME die Implementierung von Verhalten unterstützt und mit welchen Mitteln wir im Charakter-Agenten

von tinyD Verhalten realisiert haben.

Verhalten in Lime Proaktives Verhalten, Ausführen von Aktionen auch ohne dass ein äußerer Reiz vorliegt, wird durch LIME nicht explizit unterstützt. Bei der Umsetzung reaktiven Verhaltens stellt LIME mit den *Reactions* ein Konzept zur Verfügung, mit dem der Agent Veränderungen in seiner (als Tuplespaces modellierten) Umwelt registrieren sowie als Tupel gefasste Nachrichten empfangen kann (siehe Kap. 3.4.3). Die Auswertung dieser Informationen sowie die Reaktion darauf durch Aktionen des Agenten müssen selbständig entwickelt werden, ohne dass an dieser Stelle Unterstützung durch LIME-Konzepte existiert.

Verhalten des AI gesteuerten Charakter-Agenten Die Implementierung von Verhalten im Charakter-Agenten nutzt die Aufteilung des Agenten in zwei Ebenen (siehe 3.2). Der Teil des Agenten, der für die Kommunikation mit den Tuplespaces zuständig ist, benachrichtigt bei Veränderungen der Umweltsituation oder beim Empfang von Nachrichten alle angemeldeten Observer über entsprechende Methoden aus deren Interface `I_ACObserver`. Die Steuerungs-Komponente implementiert dieses Interface und ist als Observer bei der Interaktions-Komponente eingetragen, gleichzeitig hat sie über das Interface `I_ACCommandable`, das durch die Interaktions-Komponente implementiert ist, die Möglichkeit Aktionen auszuführen. Solche Aktionen können z.B. das Senden einer Nachricht oder das Wechseln des Raumes sein.

Eine AI-Steuerungskomponente ist die Klasse `CharacterCMonster.java` die von java.lang.Thread abgeleitet ist. Ihre `run()` Methode arbeitet eine Queue von Anweisungen ab, indem sie jeweils das erste Objekt aus der Queue entfernt und auf diesem die Methode `run()` aufruft. Nachdem diese abgearbeitet ist, wird dann die nächste Anweisung bearbeitet. Anweisungen sind Objekte von Klassen, die das Interface Runnable implementieren, dadurch ist sichergestellt, dass diese Objekte die Methode `run()` zur Verfügung stellen, allerdings werden diese Objekte nicht als Thread gestartet, sondern fungieren als Befehls-Objekte im Befehlsmuster. Sie werden durch diejenigen Methoden erzeugt und der Queue angefügt, die innerhalb der Verhaltenssteuerung für das reaktive Verhalten des Agenten zuständig sind. Naheliegend war, dies in den Methoden zu realisieren, die für die Implementierung des Interfaces `I_ACObserver` notwendig sind, da genau diese Methoden immer dann aufgerufen werden, wenn durch die Interaktions-Komponente Reactions ausgewertet werden.

Die jeweilige `run()` Methode der Klasse aus der das Anweisungs-Objekt

erzeugt wurde, enthält den Code, der als Reaktion ausgeführt werden soll, insbesondere werden hier die Methoden der Verhaltensausführung (Interface `I_ACCommandable`) aufgerufen, die zu dieser Reaktion gehören.

Bei einfachen Reaktionen, die nur an einer Stelle oder selten verwendet werden, bietet es sich an, die Klassen für die Anweisungs-Objekte als anonyme Klassen direkt in der Methode zu deklarieren, die das Objekt für die Queue erzeugt. Bei komplexeren Reaktionen (Übersichtlichkeit) oder wenn dieselbe Reaktion an vielen Stellen ausgelöst werden soll (Wartbarkeit) bietet es sich an eine benannte Klasse für diese Anweisung zu deklarieren. Für unseren Charakter-Agenten haben wir die erste Variante gewählt.

Eine solche reaktive Aktion kann beispielhaft der Fall sein, dass ein fremder Charakter den Raum betritt, in dem sich der Charakter-Agent zur Zeit aufhält. Alle eingetragenen Observer der Interaktions-Komponente werden mittels `updateMeetingCharacter()` benachrichtigt. Existiert zu diesem Charakter-Agenten ein Thread für Verhaltenssteuerung, dann wird bei der Abarbeitung dieser Methode ein neues Objekt erzeugt und der Queue angehängt das bei Ausführung seiner Methode `run()` einen Zeitraum von z.B. maximal 2sec. abwartet und dann mit einer Wahrscheinlichkeit von 50% den entsprechenden Charakter attackiert.

Da das Verhalten des Charakter-Agenten nicht nur reaktiv, sondern auch proaktiv sein soll, werden beim Start des Threads zur Verhaltens-Steuerung oder bei geeigneten Ereignissen Sub-Threads gestartet, die ebenfalls Objekte in die Anweisungs-Queue einspeisen. Dies geschieht beispielsweise, wenn ein Character einen Raum betritt. Bei diesem Anwendungsfall wird ein Sub-Thread erzeugt, der sich selbst für eine bestimmte Zeit schlafen legt und danach die Anweisung in die Queue legt, dass der Raum wieder verlassen wird. In dieser Zeit können beliebige andere Reaktionen ausgeführt werden.

Bei der Realisierung proaktiven Verhaltens ist zu berücksichtigen, dass diese Sub-Threads untereinander und mit dem reaktiven Programm um die Ausführung von Aktionen konkurrieren [12]. Dies wird mit steigender Zahl von proaktiven Verhaltensweisen eine geschickte Synchronisation erfordern.

Der Agent, der von einer AI gesteuert wird ist somit völlig autonom. Könnte man ihn direkt beeinflussen, widerspräche das der Idee eines autonomen Agenten. Die Möglichkeit der Übermittlung von Nachrichten zwischen Charakter-Agenten ermöglicht es, die Agenten mit einer Sprache auszustatten, die z.B. eine Kommunikation von Inhalten über die Beschränkungen der

[12] Ein Beispiel für eine solche Konkurrenz ist der Fall, dass das proaktive Programm ein verlassen des Raumes in 24sec vorsieht, das reaktive Programm aber ein sofortiges Verlassen des Raumes anstößt. In diesem Fall darf und kann das proaktive Programm hier nicht mehr tätig werden.

Interfaces hinaus ermöglicht. Ein autonomer Agent, kann so mit Hilfe von an ihn adressierten Nachrichten zu jeglichem Verhalten gebeten werden, sofern seine Implementierung die Nachricht entsprechend interpretieren kann. Für ein konkretes Beispiel siehe Abschnitt 3.3.3.

3.3.2 Benutzersteuerung

Die Steuerung von Charakter-Agenten kann mit Hilfe einer GUI durch einen menschlichen Spieler erfolgen, in diesem Fall übernimmt dieser die Auswertung von Informationen und trifft die Entscheidung über Aktionen.
Die GUI von tinyD enthält eine Vielzahl von Funktionen, die hier nicht alle beschrieben werden sollen, da sie für die Bewertung von LIME nicht relevant sind. Grundlegend ist das Konzept der GUI, für jeden Charakter-Agenten ein Panel zur Verfügung stellen zu können. Ein solches Panel kann als Observer oder Commander bei einem Charakter-Agenten angemeldet sein. Es enthält stets Anzeigen für Attribute, Position etc. Ist das Panel als Commander angemeldet, enthält es zusätzlich Kontrollelemente, die dem Benutzer Raumwechsel, sprechen, flüstern, angreifen usw. ermöglichen.
Durch die Benutzung beliebig vieler Panels kann der Benutzer gleichzeitig mehrere Charakter-Agenten steuern und beobachten. Ihm stehen ferner Dungeon-Maps zur Verfügung, auf denen die aktuellen Positionen der kontrollierten und der beobachteten Charakter-Agenten angezeigt werden.
Auch ist ein Kontrollfenster verfügbar, welches alle auf dem physikalischen Host präsenten Charakter-Agenten anzeigt und manipulieren lässt (z.B. Aufschalten von AI oder Initialisierung nicht initialisierter Charakter-Agenten).

3.3.3 Wechsel der Steuerungsart

Ein Charakter-Agent führt, solange er keine Anweisung eines Commanders bekommt, keine Aktionen aus. Genau ein Commander kann bei einem Charakter-Agenten angemeldet sein. Der angemeldete Commander hat dann die exklusive Steuerung über den Charakter-Agenten.
Ist der angemeldete Commander die GUI eines Benutzers, so kann der Benutzer die Steuerung durch Wahl des Menüpunkts “release control“ wieder abgeben und ein neuer Commander kann sich anmelden.
Ist eine AI als Commander angemeldet, wird sie die Steuerung nicht von selber abgeben. Die Abgabe der Steuerung direkt bewirken zu können, widerspräche dem Autonomiegendanken eines AI gesteuerten Agenten. Des-

wegen wird hierzu die Kommunikationsmöglichkeit zwischen Agenten durch Nachrichten genutzt.
Unsere Implementierung des Charakter-Agenten kann per Nachricht "gebeten" werden die Steuerung abzugeben. Dazu muss dem Charakter-Agenten im Spiel lediglich die Nachricht LEASE CONTROL übermittelt werden. Dies wird von unserer Implementation verstanden und bewirkt, das sich der Verhaltenssteuerungs-Thread beendet. Damit ist die Übernahme der Steuerung durch einen anderen Commander (GUI oder andere AI) wieder möglich.

3.4 Verwendung und Bewertung von Lime Konzepten

Bei der Implementierung von tinyD haben wir verschiedene Konzepte von LIME verwendet. Die dabei gemachten Erfahrungen werden im folgenden vorgestellt.

3.4.1 Tuple und Templates

Tuple sind zentraler Bestandteil von LIME, wir haben sie sowohl verwendet um Nachrichten zwischen Agenten auszutauschen, als auch Umwelt-Situationen zu beschreiben (z.B. Räume und Verbindungen). Die Implementierung der Tuple, wie sie in LIME realisiert ist, entspricht zwar ganz den *Linda* Vorgaben, allerdings ist der Umgang mit diesen sowohl umständlich als auch fehleranfällig. Um in LIME ein Tuple oder ein Template zu definieren, muss zunächst ein Tuple-Objekt angelegt werden, danach müssen diesem sukzessive Felder entweder als konkrete Werte (*actual*) oder als Definition eines Typs (*formal*) hinzugefügt werden.

Soll auf diese Felder im Weiteren wieder zugegriffen werden, entweder um Werte abzuändern oder um sie auszulesen, dann muss der korrekte Index des Feldes verwendet werden. Wird ein bestimmter Tupel-Typ an verschiedenen Stellen im Code zusammengesetzt, dann muss die Reihenfolge der Felder exakt gleich sein, dies ist sehr fehleranfällig und erfordert von den Entwicklern eine hohes Maß an Sorgfalt. Änderungen in der Reihenfolge der Felder erfordern im Worst Case ein Durchsuchen des gesamten Codes nach der Verwendung dieses Tuples. Dies macht das Programmieren mit LIME-Tupeln fehleranfällig. Da Tuple oder Templates für jeden Kommunikationsvorgang oder jede Interaktion mit der Umwelt des Agenten verwendet werden, war dies ein nicht unwesentliches Problem.

Um dem zu begegnen, haben wir eine Factory (Klasse: `tuples.HolderFactory`) entwickelt, die ausgehend von der Bezeichung

des Tuples oder einem konkretem Tuple, aus einer zentralen Sammlung (Klasse: `tuples.DefaultTupleSpec`) von Tupledefinitionen (Klasse: `tuples.TupleSpec`) einen Wrapper (Klasse: `tuples.TupleHolder`) erzeugt, der ein passendes Tuple (bzw. Template) enthält. Dieser Wrapper verfügt über Methoden, Felder des Tuples auszulesen und zu schreiben, wobei das jeweilige Feld als String Parameter angegeben wird.

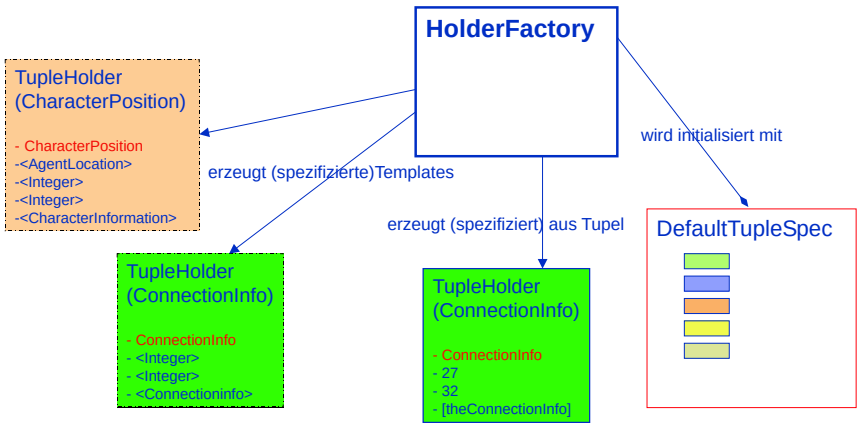

Abbildung 4: Schematische Darstellung des TupleHolder-Konzeptes

Wird ein TupleHolder zu einem konkreten Tuple erzeugt, dann sind über diesen Wrapper alle Werte von im Tuple mit *acutals* belegten Feldern über die Bezeichnung (als String aus der zentralen Tupledefiniton) des Feldes abrufbar(`myTupleHolder.getValue(,fieldname‘)`). Der TupleHolder enthält also ein schon vor-initialisiertes Tuple. Wird der Wrapper nicht zu einem konkreten Tuple erzeugt, sondern durch den Aufruf `HolderFactory.createTupleHolder(,tuplename‘)`, dann enthält der Wrapper ein Tuple, dessen Felder entsprechend der Tupledefinition zu `tuplename` mit *actuals* und *formals* vorinitialisiert sind. Wrapper und Factory bieten zum Umgang mit Tupeln einige Vorteile:

- Die Erzeugung von Tuplen mit vorinitialisierten Werten ist einfacher und sicherer.
- Das Auslesen und Bearbeiten von Werten wird durch den Wrapper gekapselt.

- Durch die Verwendung von `setValueSafe()` statt `setValue()` kann beim Manipulieren von Werten Typsicherheit zur Laufzeit garantiert werden (bei Fehler kontrollierte Exception).

- Durch die Verwendung von Konstanten als Bezeichner für Tupledefintion und Feldnamen können weitere Fehler vermieden werden.

- Tupledefinitionen sind skalierbar in dem Sinne, dass ihnen beliebig neue Felder hinzugefügt werden können, der auf der alten Tupledefinition basierende Code aber dennoch lauffähig bleiben kann (Abwärtskompatibel).

Es ist möglich, die Factory so zu erweitern, dass sie auch aus einem Objekt, durch den Aufruf einer entsprechenden Methode, Tupel erzeugen kann, die dann schon mit den Daten dieses Objektes gefüllt sind. Dazu könnten die Klassen solcher Objekte mit einer geigneten Factory-Methode ausgestattet sein und ein gemeinsames Interface implementieren in dem diese Methode deklariert ist.

Ein interessanter Aspekt bei der Verwendung von Tuplen in LIME ist die Frage, wie ein Tuple aufgebaut werden sollte. Ist es sinnvoll, dass Umweltdaten oder Nachrichten, die im Agentensystem zunächst als Objekt vorliegen, in ein Tuple exportiert werden? Dann ist zu überlegen, ob das Objekt als *ein* Feld in das Tuple eingefügt wird oder ob alle relevaten Attribute des Objektes ausgelesen und einzeln in Felder des Tuples abgelegt werden oder ob eine Mischform aus beiden Varianten verwendet wird.

Es hat sich gezeigt, dass der einfachste Umgang mit dieser Frage ist, das Objekt komplett im Tuple abzulegen und Attribute des Objektes, die zur Identifizierung des Tuples über geeignete Templates benötigt werden, zusätzlich in Felder im Tupel abzulegen. Beim Matching wird das Template dann nur hinsichtlich der zusätzlich exportierten Attribut-Felder spezifiziert, beim Zugriff auf die Daten wird dagegen das vollständige Objekt genutzt.

Problematisch ist in diesem Zusammenhang, dass beim Schreiben des Tuples in den Tuple-Space sowie beim Lesen desselben eine Kopie mittels eines inLIME eingebauten vollständigen Clonings, von allen Feldern angelegt werden. Handelt es sich bei den Objekten der Felder um komplexe Objekte mit umfangreichen Referenzen, dann wird mit jedem erzeugten Tuple ein komplettes Objektsystem geklont. Sollen Daten aus einem komplexen Objektsystem über einen TupleSpace übertragen werden und danach wieder in ein ähnliches Objektsystem integriert werden, dann müssen die Referenzen explizit nach-gepflegt werden, hier bietet LIME keine Transparenz, ist also

kein verteiltes Objektsystem. Wir haben dieses Problem durch die Verwendung von einfachen Informations-Klassen (wie z.B. durch Verwendung von `DungeonInformation` bei Tuplespace Transaktionen anstelle der komplexen Klasse `Dungeon`) gehandhabt, die keine Referenzen enthalten, dadurch wird der Aufwand durch Klonen von Objekten gering gehalten. Allerdings entsteht zusätzlicher Aufwand durch die Erzeugung und Verwertung der Informations-Objekte.

Da Tupel bei ihrer Übertragung serialisiert werden, muss beachtet werden, dass verschiedene Java Versionen für inkompatible IDs der serialisierten Objekte sorgen, das System also dann nicht mehr funktionieren wird. Allerdings kann diese ID explitzit überschrieben werden.

3.4.2 Tuplespace

Bei der Verwendung der LIME Tuplespaces fiel auf, dass der Zugriff auf diese nur dem Thread gestattet ist, der den Tuplespace erzeugt hat. Da sowohl GUI als auch AI eigenständige Threads sind, der Tuplespace für die Spielumgebung aber vom Thread des LIME Agenten erzeugt wurde, können diese keine Methoden auf dem Charakter-Agenten aufrufen, die direkt Manipulationen an einem Tuplespace vornehmen. Stattdessen wird ein Befehls-Objekt in eine dafür vorgesehene Queue des Charakter-Agenten eingestellt, das dann von dessen Thread aufgerufen wird. Dieses Verfahren ist aber in dem Fall problematisch, wenn auf dem LIME-Agenten eine get-Methode aufgerufen wird, die eine Tuplespace-Operation ausführen muss. In diesem Fall muss die get-Methode den aufrufenden Thread natürlich so lange blockieren, bis das Befehlsobjekt einen Wert als Ergebnis zurückliefert. Wir haben dieses Problem mit einer Monitor-basierten Thread-Synchronisation gelöst, bei welcher der aufrufende Thread als Monitor fungiert, der sich selbst im Code der get-Methode schlafen legt und vom Befehlsobjekt per `notify()` wieder aufgeweckt wird.

Problematisch ist bei den LIME Tuplespaces, dass der Zugriff auf diese nicht durch Zugriffkontrolle eingeschränkt ist, d.h. jeder Agent, der den Namen eines Tuplespaces im System kennt, kann einen eigenen Tuplespace dieses Namens erzeugen, auf diesem `share` aufrufen und die Daten aller shared Tuplespaces dieses Namens zugreifen. Alle Daten in shared Tuplespaces sind also prinzipiell öffentlich, geschützte Kommunikation erfordert zumindest eine geschützte Einigung über zu verwendende Tuplespace Namen. Hier ist es möglich mit Public-Key Verfahren zu arbeiten. In jedem Fall verdient dieser Punkt bei Anwendungen mit sensiblen Daten besondere Beachtung. Lime jedenfalls bietet hier keine explizite Unterstützung.

3.4.3 Reactions und Verhalten

Die Benutzung von Reactions ist prinzipiell problemlos, trotzdem gibt es einige Aspekte, die zu beachten sind.

- Bereits existerende Tupel im Tuplespace, die zum Template einer Reaction passen, lösen diese aus, wenn die Reaction auf den Tuplespace gebunden wird - auch wenn die Tupel zeitlich vor der Bindung der Reaction erzeugt wurden.

- Bei der Entwicklung der Templates für verschiedene Reactions muss sorgfältig gearbeitet werden. Problematisch sind hier insbesondere Templates, die Untermengen von Tupeln anderer Templates definieren. Wird ein solches Tuple in den Tuplespace eingestellt, dann werden zwei Reactions ausgelöst. Dies ist bei der Auswertung der Reactions zu beachten.

- Der Umgang mit der Reaction bei Ihrer Behandlung im Code des Agenten wird anhand des auslösenden Tuples bestimmt. Das Tuple muss dazu so definiert sein, dass es einen eindeutig abgrenzbaren Sachverhalt beschreibt, anhand dessen eine Reaktion erfolgen kann. Im Zweifel muss diesem Tupel ein geeignetes Zustands-Feld angefügt sein (Beispielsweise ein kennzeichnender String: ‚Nachricht', ‚Verbindung zu anderem Raum'o.ä.), dass dem Agenten einen Rückschluss auf notwendige Maßnahmen erlaubt.

Ein Vorteil des Reaction Konzeptes ist, dass jeder Agent die für ihn relevanten Informationen über geeignete Templates selbst filtern kann, vorausgesetzt, die Tupel haben entsprechende Attribute. Was allerdings fehlt, ist die Möglichkeit eine Reaction zu definieren, die auf alle neuen Tupel eines Templates, aber *nicht* auf die eines (oder mehrer) anderen Templates (Sub-Menge) reagiert. Dies könnte es leichter machen, unerwünschte Daten schon vor Auslösen der Reaction abzufangen[13].

3.4.4 Interaktion mit Lime

Die Kombination der Konzepte Tuple, Tuplespace sowie Reaction lassen sich gut für den Austausch von Nachrichten zwischen Agenten nutzen. Einfach ist es damit, asynchrone Kommunikation zu realisieren, zeitliche wie räumliche Asynchronität sind grundlegende Eigenschaften des *Linda*-Konzeptes. Allerdings waren die asynchronen Konzepte im wesentlichen bei der Publizierung

[13] Ein Agent könnte so beispielsweise Reaktionen auf eigene Tupel herausfiltern

von und Suche nach Diensten (in unserem Falle Spiele) von Nutzen, in vielen anderen Fällen musste die Asynchronität durch Handshake Protokolle mit Timeout begrenzt werden, wobei der Kommunikationsvorgang entweder blockierend sein konnte oder der Agent weiterhin noch Handlungsfähig bleiben musste. Blockierendes Warten auf Nachrichten wurde realisiert durch aktives Warten (mit `Thread.wait(time)`, gefolgt von einem `inp(template)` - als Schleife), nichtblockierendes Warten durch das Registrieren von neuen Reactions auf das interessante Template und Behandlung der Nachricht in der `reactsTo()` Methode des Agenten. Die Verwendung von `in(template)` bot sich nur selten an, wenn kein timeout der blockierenden Wirkung nötig war.

Um die Komplexität des Systems gering zu halten sollte bei der Definition der Tuple für Kommunikation versucht werden, für einen (mehrphasigen) Kommunikationsvorgang genau eine Tupeldefinition zu verwenden und den Status der Verhandlung über ein Feld im Tupel zu markieren (z.B. Status: {Request, Offer, EngageRequest, EngageApproved})

3.4.5 Mobilität und Multi-Host Systeme

LIME selbst stellt keine Unterstützung für mobilen Code bereit, allerdings wird μCODE als Add-On angegeben, mit dem eine Migration von LIME Agenten möglich sein soll. Wir haben versucht, eine physische Migration von laufenden Agenten zwischen Java VMs damit zu realisieren. Migration erfordert allerdings, dass alle zur Ausführung des Agenten benötigte Klassen mit dem Agenten migriert werden. Der von μCODE verwendete Mechanismus, diese benötigten Klassen zu identifizieren hatte aber Schwierigkeiten mit den Objekten vom Type `Class`, die per `Class.forName()` erzeugt wurden. Wir haben uns deshalb auf die Realisierung von logischer Migration, d.h. der Möglichkeit für Agenten auf entfernten Host zu spielen, beschränkt. Dies war mit den Mitteln von LIME auch realisierbar und erfüllt die Forderung nach einer physischen Mobiltät der Hosts insoweit, als das nach der Aufnahme des Hosts in ein ad-hoc Netzwerk eine Teilnahme der Agenten dieses Hosts an Spielen im Netz möglich ist.

Da jeder Agent auf den `LimeSystemTuplespace` Reactions auf das Hinzukommen oder Wegfallen von Hosts, Agenten oder Tuplespaces registrieren kann, ist eine dezentrale Verwaltung eines verteilten Agenten-Systemes realisierbar. Wir haben dies z.B. genutzt, um automatisch Verbindungen zwischen Spielsystemen auf- und abzubauen.

Als Problematisch anzusehen ist allerdings die Stabilität eines LIME Systems aus mehreren Hosts. Fällt ein Agent oder ein Host aus bzw. reagiert

dieser nicht mehr, dann bringt dies in vielen Fällen das gesamte System zum Absturz. Dazu kommt, dass ein solcher Absturz eines Agenten immer wieder mal spontan auftritt. In einem verteilten Agenten-System ist eine integrierte Fehlerbehandlung solcher Fälle durch LIME eigentlich unverzichtbar.

4 Zusammenfassung

Die Umsetzung eines agentenbasierten Spiels mit Hilfe von LIME hat Stärken und Schwächen sowohl des dahinter stehenden Konzeptes, als auch des Systems selbst aufgezeigt. Die Vorzüge von LIME zusammenfassend kann an erster Stelle genannt werden, dass die einfachen Kommunikationsmechanismen eine schnelle Entwicklung einfacher Agenten ermöglicht. Das *Linda* Kommunikationsmodell bietet vor allem bei der Realisierung von zeitlich und räumlich asynchroner Kommunikation anonymer Peers Vorteile, bei Agenten-Systemen kann dies vor allem bei der Publizierung und der Suche nach allgemein verfügbaren Diensten oder Informationen genutzt werden. Die Simulation anderer Interaktionsparadigmen erfordert die Erweiterung der verwendeten Tupel um Informationen wie Adressat, Gültigkeitsdauer o.ä. und gegebenenfalls die Synchronisation der Kommunikation durch spezielle Tupel. Die Handhabung und Wartung solcher komplexen Tupel erfordert entweder ein sehr diszipliniertes Programmier-Verhalten oder die Entwicklung von Hilfsmitteln um die Komplexität beherrschbar zu machen. Ähnlich sieht es bei der Entwicklung komplexer Agenten aus, insbesondere bei der Implementierung von Verhalten. Die Komplexität der Agenten steigt mit der Zahl an Zuständen und der Zahl beteiligter paralleler Threads beträchtlich an, so dass auch hier Konzepte auf höherer Systemebene erforderlich werden. Kritisch anzumerken ist, dass die Abstraktion von der physischen Ebene des Agenten-Systems zum einen nicht vollständig ist (vgl. Locations), zum anderen aber auch insbesondere in der Fehlerbehandlung nicht verlässlich ist. Verbesserungswürdig ist weiterhin die Organisation der am System teilnehmenden Hosts, sowohl das Konzept des im vorhinein festzulegenden Leaders, als auch die Beschränkung auf LAN sind problematisch.

Fazit Wie kann nun die Eignung von LIME als Framework für Multi-Agenten-Systeme beurteilt werden? LIME zeigt, dass das Shared-Memory Konzept von *Linda* sich für die Entwicklung von Agenten-Systemen prinzipiell eignet und insbesondere eine schnelle Entwicklung einfacher und offener Systeme fördert. Bei der Realisierung komplexerer Systeme tritt aber die Bereitstellung von Unterstützung für die Realisierung von Verhalten und

Kommunikation auf höherer Ebene in den Vordergrund und hier bietet LIME keine Unterstützung. Bei der Entwicklung von Agenten-Systemen sollte der wesentliche Entwicklungs-Aufwand auf das Verhalten der Agenten entfallen, bei der Verwendung von LIME ist aber zunächst schon der Aufwand zur Realisierung von Kommunikation auf unterer Ebene noch relativ hoch. Hinsichtlich des Laufzeit-Systems kann festgestellt werden, dass Stabilität und Behandlung von Ausnahmen oder Fehlern in LIME nicht ausreichend sind um damit Anwendungen zu entwickeln, die höhere Anforderungen an diese stellen, als dies bei einem Spiel der Fall ist. Dennoch ist LIME als System immerhin so reif, dass die Entwicklung eines Agenten-basierten Spieles damit möglich war.

Literatur

[1] Gian Pietro Picco, Amy L. Murphy, Gruia-Catalin Roman. *Lime: A Middleware for Physical and Logical Mobility*. In Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21), Phoenix, Arizona, USA, April 2001.

[2] Gian Pietro Picco, Amy L. Murphy, Gruia-Catalin Roman. *Developing Mobile Computing Applications with Lime*. In Proceedings of the 22th International Conference on Software Engineering (ICSE 2000), Limerick (Ireland), M. Jazayeri and A. Wolf eds., ACM Press, Juni 2000.

[3] Gian Pietro Picco, Amy L. Murphy, Gruia-Catalin Roman. *Lime: Linda Meets Mobility*. In Proceedings of the 21st. International Conference on Software Engineering (ICSE'99), Los Angeles (USA), Mai 1999.

[4] Amy L. Murphy. *Enabling the Rapid Development of Dependable Applications in the Mobile Environment*. Washington University Technical Report WUCS-00-15, Washington, August 2000.

[5] Gian Pietro Picco. *Understanding, Evalutaion, Formalizing, and Exploiting Code Mobility*. Doktorarbeit, Politecnico di Torino, Italien, Februar 1998.

[6] George Coulouris, Jean Dollimore, Tim Kindberg. *Distributed Systems - Concepts and Design*. Addison-Wesley, ISBN 0201619180, Amsterdam, 2001.