Seminar-Arbeit

Im Rahmen des Seminars „Verteiltes Programmieren von Agenten Systemen“

zum Thema

Erstellung eines Multi-Agenten-Systems mit Hilfe von LIME

Themensteller:

Prof. Guido Wirtz

Dr. Dietmar Lammers

vorgelegt von:
Jens Bruhn, Sven Kaffille, Marcel Sieke

Münster SoSe 2002

Inhaltsverzeichnis

Abbildungsverzeichnis **II**

Abkürzungsverzeichnis **III**

1 Einleitung **1**

2 Lime im Überblick **2**

2.1 Linda-Tuplespace 2

2.2 Umsetzung in Lime 3

2.2.1 Agent 4

2.2.2 Besonderheiten bei Operationen 5

2.3 Probleme 6

2.4 Bewertung 6

3 Das Casino-Konzept **8**

3.1 Zielsetzung 8

3.2 Typsicherheit 8

3.3 Umgebungen 9

3.4 Das Plugin-Konzept 11

3.5 Spielleiter 13

3.6 Spieler 14

3.6.1 Computerspieler 14

3.6.2 Menschliche Spieler 14

3.7 Migration 14

3.8 Observer 16

3.9 Das Spiel „Meier“ 17

3.9.1 Spielregeln 17

3.9.2 Das GUI des Spiels 18

4 Starten des Systems **20**

5 Zusammenfassung und Ausblick **22**

Literatur- und URL-verzeichnis **26**

Literatur 26

URLs 26

Abbildungsverzeichnis

Abbildung 1: Aufbau eines Agenten 12
Abbildung 2: GUI des Observers 16
Abbildung 3: GUI des Spielleiters. 18
Abbildung 4: GUI eines Computerspielers. 19
Abbildung 5: GUI für einen menschlichen Spieler. 19

Abkürzungsverzeichnis

Bspw.	beispielsweise
Etc.	et cetera
f.	folgende
ff.	fortfolgende
MAS	Multi-Agenten-System
S.	Seite
Vgl.	Vergleiche
TS	Tuplespace
z.B.	zum Beispiel

1 Einleitung

Im Rahmen des Seminars „Verteiltes Programmieren von Agenten-Systemen“ bestand die Aufgabe darin, ein Agenten-System mit Hilfe eines von den Autoren auszuwählenden Tools zu implementieren und die Erfahrungen mit diesem Tool sowie das implementierte System zu dokumentieren.

Ein Agent[1] ist ein autonomes, intelligentes Stück Software, welches im Interesse eines ‚Users’ eine bestimmte Aufgabe erledigt. Agenten kommunizieren untereinander und kooperieren unter Umständen, um ihre Aufgaben zu erfüllen. Die Aufgabe der implementierten Agenten ist die selbständige Austragung eines Spiels. Darüber hinaus sollen die Agenten fähig sein, sich in unterschiedlichen logischen und physischen Umgebungen zu bewegen, was bedeutet, dass die Agenten die Fähigkeit erhalten unterschiedliche Spiele zu spielen (wechselnde logische Umgebungen) und von einem Rechner zu einem anderen Rechner im Netzwerk zu migrieren (ändern der physischen Umgebung).

In der vorliegenden Arbeit wird zunächst im Kapitel 2 auf das ausgewählte Tool – das Lime-Framework[2] – als Grundlage für das entwickelte Multi-Agenten-System eingegangen. Hierbei wird zunächst auf das zugrunde liegende Konzept des *Linda-Tuplespace* eingegangen, um anschließend die Spezifika der Implementierung in Lime zu untersuchen. Die hiermit im Zusammenhang stehenden Probleme werden im Kapitel 2.3 erläutert. Den Abschluss bildet eine Bewertung des Lime-Frameworks. Hierbei wird vor allem auf eine mögliche Anwendung im Rahmen von Agentensystemen eingegangen.

Kapitel 3 stellt kurz die Zielsetzung der Autoren dar und beschreibt anschließend die verwendeten Konzepte sowie die Bestandteile des Systems. Auf Implementierungsdetails wird hierbei verzichtet.

Im vierten Kapitel wird beschrieben, wie das entwickelte System gestartet wird, wonach in Kapitel 5 eine kurze Zusammenfassung und ein Ausblick auf Erweiterungen des implementierten Systems erfolgen.

---

[1] Vgl. O’Brien, P. D., Nicol, R. C. (1998), S. 51.

[2] http://lime.sourceforge.net.

2 Lime im Überblick

2.1 Linda-Tuplespace

Wie einleitend bereits erwähnt stellt Lime eine Implementierung des Linda-Tuplespace-Konzeptes dar. Linda stellt ein Shared-Memory-Konzept für ein verteiltes System dar. Grundlage dieses Konzeptes ist der so genannte *Tuplespace (kurz: TS)*. Ein TS ist ein im System globaler Speicher, auf dem Programme auf den einzelnen Rechenknoten lesen und schreiben können. Er verfügt im Wesentlichen über die Eigenschaften Persistenz und Knotenunabhängigkeit.

Persistenz: Daten, die in einen TS geschrieben wurden, verbleiben in diesem, bis sie explizit wieder gelöscht werden, d.h., dass sie bei einem Systemneustart wieder vorhanden sind. Ein TS kann also nicht nur zur Kommunikation, sondern auch als persistenter Speicher verwendet werden.

Knotenunabhängigkeit: Ein TS ist nicht an einen bestimmten Rechenknoten gebunden. Wenn z.B. ein Knoten Daten in einen TS schreibt und anschließend das System verlässt, so sind die Daten weiterhin verfügbar. Auch sind die TSs nicht von Abstürzen einzelner Knoten beeinträchtigt.

Daten werden in einem TS als Tupel repräsentiert. Ein Tupel besteht aus mehreren Feldern, deren Anzahl und Datentypen vom Anwender frei wählbar sind. Um Lese- oder Schreiboperationen auf einem TS auszuführen, reichen die drei Grundoperationen *out*, *in* und *read* aus, sofern die Operationen in und read nicht blockierend sind. Mit *out* wird ein Tupel in einen TS geschrieben, mit *in* wird es wieder entfernt und mit *read* lediglich gelesen, wobei eine Kopie des Tupels im TS verbleibt. Passen mehrere Tupel zu einer Anfrage, so wird eines dieser Tupel zurückgegeben, wobei gemäß der Spezifikation keine Regel vorgegeben ist, nach der eines der möglichen Tupel ausgewählt wird. Um einen komfortableren Umgang mit Linda zu ermöglichen, kommen blockierende *read*- und *in*-Operationen, sowie Operationen zum Lesen und entfernen aller zutreffenden Tupel hinzu. Die Auswahl eines Tupels beim Lesen und Entfernen erfolgt über ein Template, d.h. der Anwender spezifiziert, welche Felder das von ihm gewünschte Tupel beinhalten soll und ggf. ihren Inhalt.

2.2 Umsetzung in Lime

Die Umsetzung des Linda-Konzeptes in Lime weicht in einigen zentralen Punkten von der Grundkonzeption ab. Im Wesentlichen wurden die Operationen und die Idee des TS beibehalten.

Zentrale Komponente bei der Umsetzung in Lime ist der so genannte *LimeServer*. Er stellt die Kommunikationsschnittstelle für Agenten mit der Umgebung des Servers dar. Einer der LimeServer des Systems muss als *Leader* für das gesamte Netz deklariert werden. Dieser übernimmt die Koordination zwischen den einzelnen beteiligten Servern. Wird ein Server gestartet, so muss er, um mit anderen Servern interagieren zu können, *engaged* werden. Bis zu diesem Zeitpunkt koordiniert er lediglich die einzelnen Agenten auf dem Server und interagiert nicht mit anderen Servern. Agenten werden in einem Server gestartet, d.h. sie werden unter der Kontrolle des Servers.

TSs werden in Lime in *shared* und *unshared* unterteilt. Grundsätzlich ist jeder TS bei seiner Erstellung unshared, d.h. er ist lediglich vom erstellenden Agenten aus zugreifbar und für den Rest des Netzwerkes unsichtbar. Wird ein TS geshared, so können andere Agenten, die ebenfalls einen TS mit gleichem Namen geshared haben, auf den neuen Teil-TS zugreifen. Hieraus ergibt sich eine grundlegende Abweichung vom Linda-Konzept. Es existieren nicht nur globale TSs, sondern ein TS setzt sich aus den einzelnen shared TSs der Agenten zusammen. Setzt ein Agent seinen Teil-TS auf unshared oder verlässt er das System, so kann auf Daten dieses Teils des globalen TSs nicht mehr zugegriffen werden. Somit ist ein Lime-TS nicht ortsunabhängig und auch nicht persistent. Ein TS in Lime ist folglich nicht mit einem TS bei Linda gleichzusetzen. In Lime wird ein TS als eine Art Koordinations- und Kommunikationsplattform für Agenten betrachtet. Auf einem TS sind die oben beschriebenen sieben Operationen möglich, wobei bei den nicht blockierenden Operationen die Ortsabhängigkeit der Teil-TSs zu beachten ist. Dies wird weiter unten genauer erläutert.

Zum Ablegen von Daten in den TSs verwendet Lime die Tupel-Klassen der Linda-Implementierung *Lights*[3], von der Lime im Wesentlichen eine Erweiterung um die verteilte Anwendung in einem Netzwerk darstellt. Auf Lights soll an dieser Stelle nicht genauer eingegangen werden. Wichtig ist allerdings, dass Lights keine verschiedenen Tupel-Typen erlaubt. Das Matching zwischen Tupel und Template erfolgt also ausschließlich über die

einzelnen Einträge. Hierbei muss für jeden Eintrag explizit der Typ und ggf. der gewünschte Inhalt angegeben werden. Ferner wird bei Tupel-Feldern, deren Inhalt nicht spezifiziert wird, zwischen *formal*- und *null*-Feldern unterschieden. Formal-Felder dienen zur Suche von Tuplen, wenn der konkrete Wert des Eintrages für die Suche nicht relevant ist. Sie werden folglich im Zusammenhang mit Templates eingesetzt. Null-Felder werden verwendet, wenn ein Eintrag eines Tuples nicht gesetzt werden soll. Diese beiden Feldertypen sind erforderlich, da Lime beim Vergleich zweier Tupel die Reihenfolge der Einträge beachtet. Die einzelnen Felder sind nur anhand ihrer Position im Tupel identifizierbar und nicht über einen Schlüssel oder ähnliches. Diese Implementierung stellt zwar keine grundlegende Abweichung von Linda dar, verhindert aber eine Typsicherheit bei Tupeln, d.h. alle Tupel sind vom gleichen Typ und es kann z.B. nicht explizit nach einem Tupel vom Typ „Spielankündigung“ gesucht werden. Es ist zwar möglich z.B. den Typ des Tuples durch einen String-Eintrag zu kennzeichnen, um ihn von anderen Tuplen mit gleicher Felderanzahl und gleichen Datentypen der Einträge zu unterscheiden, eine Veränderung der Länge des Tuples oder der Datentypen würde aber dazu führen, dass Agenten, die Tupel der alten Struktur suchen, neue Tupel nicht finden würden.

Neben den üblichen Operationen wie blockierendes Lesen, Schreiben etc. bietet Lime auch noch die Möglichkeit sich mit Hilfe von Events, so genannten *Reactions*, über das Eintreffen eines Tupels in einem bestimmten TS informieren zu lassen. Dazu muss sich der Agent beim TS für das gewünschte Ereignis registrieren und eine Instanz einer Klasse bereitstellen, welche die Benachrichtigung über das Event entgegennimmt. Diese Instanz kann der Agent selber oder eine andere Klasse sein, die das Interface `lime.ReactionListener` implementiert.

2.2.1 Agent

In Lime werden zwei Arten von Agenten unterstützt. Die erste Gruppe bilden die *StationaryAgents*, welche als Beteiligte immer in einem LimeServer ausgeführt werden. Die zweite Gruppe der *MobileAgents* ist durch ihre Migrationsfähigkeit gekennzeichnet. Hierbei wird auf *µCode*[4] zurückgegriffen. Da aber im Rahmen der Implementierung eine Verwendung

---

[3] http://lights.sourceforge.net.

[4] http://mucode.sourceforge.net.

durch etliche Probleme nicht sinnvoll erschien, wurde zur Migration ein anderes Verfahren verwendet. Hierbei war es möglich StationaryAgents migrieren zu lassen.

Alle Agenten, welche einen TS erstellen und auf diesen zugreifen möchten, müssen von der Klasse `lime.StationaryAgent` erben. Ein Operationsaufruf von einem Agenten A (oder einer anderen Instanz einer Java-Klasse) auf die von einem Agenten B erzeugte Instanz eines TS ist nicht möglich. Soll bspw. aus einer GUI ein Tupel aus einem TS ausgelesen werden, hat diese nicht die Möglichkeit die entsprechende in- oder inp-Methode des TS aufzurufen, obwohl sie eine Referenz auf den TS besitzt. Nur die Thread, welche zu einem StationaryAgent gehört und den TS erstellt hat kann Methoden auf diesem ausführen, da durch den LimeServer die ThreadID der Threads der eine Operation auf dem TS ausführen möchte mit der ID des Threads, die den TS erzeugte, verglichen wird. Stimmen diese nicht überein wird eine Exception ausgelöst. Soll es z. B. einem menschlichen Spieler mittels einer GUI ermöglicht werden Tupel aus einem TS zu entnehmen bzw. Tuple in einen TS zu stellen, muss diese die Möglichkeit haben dem Agenten, welcher den TS erzeugt hat, den hierzu nötigen Auftrag zu erteilen. Der Agent führt dann die entsprechende Operation in seiner run()-Methode stellvertretend für die GUI aus und teilt dieser dann auch das Ergebnis mit. Durch dieses Vorgehen bleibt der erzeugende Agent immer ausführende Thread bezogen auf den jeweiligen TS und es kann nicht zu unerwünschten Fehlermeldungen kommen.

2.2.2 Besonderheiten bei Operationen

Wie oben bereits erwähnt, ist ein TS in Lime nicht ortsunabhängig. Hieraus resultiert eine Besonderheit von Lime. Bei allen Operationen besteht die Möglichkeit anzugeben, in welchen Teil-TS ein neues Tupel geschrieben werden soll bzw. aus welchem Teil-TS ein Tupel gelesen werden soll. Für alle nicht blockierenden Operationen ist die Angabe der *Location* zwingend vorgegeben, aus der das jeweilige Tupel bezogen werden soll. Das Gleiche gilt für die Operationen, mit denen Gruppen von Tupeln gelesen bzw. entfernt werden können. Laut Spezifikation wird beim Schreiben ohne Angabe einer Ziel-Location das jeweilige Tupel in keinen bestimmten Teil-TS geschrieben. Soll also ein Tupel ggf. mit einer nicht blockierenden Operation gelesen werden, so muss beim Schreiben des Tuples die Location mit angegeben werden. Es ist für jeden Agenten möglich Tupel in jeden beliebigen Teil-TS zu schreiben. Daraus folgt, dass lediglich die Operation in, zum entfernen von Tuplen, ohne Ortsangabe auskommt. Bei den Locations wird zwischen *AgentLocation* und *HostLocation* unterschieden. Beim Schreiben von Tupeln ist lediglich die AgentLocation anzugeben, wenn das Tupel in einen speziellen Teil-TS geschrieben werden soll. Beim nicht-blockierenden

Lesen bzw. Entfernen, muss mindestens eine Location angegeben werden, in der gesucht werden soll. Die Angabe einer zweiten Location ist optional. Es werden lediglich die angegebenen Locations durchsucht.

2.3 Probleme

Die im vorigen Abschnitt beschriebene Vorgehensweise von Lime bei einigen Schreib- und Lese-Operationen führt dazu, dass Agenten, die nicht durch in-Operationen blockiert werden wollen, die *Location* ihrer Kommunikationspartner kennen müssen bzw. diese vor Aufnahme der Kommunikation ermitteln müssen.

Neben den bereits beschriebenen Schwierigkeiten und Besonderheiten von Lime ist vor allem das Sharing und Unsharing von TSs als sehr problematisch anzusehen. Wird Lime in einem Kontext eingesetzt, in dem im Zeitablauf Agenten das System verlassen und gleichzeitig neue Agenten hinzukommen, so kann das Phänomen auftreten, dass das gesamte System stehen bleibt und sich nicht zur Wiederaufnahme der Tätigkeit bewegen lässt.

Ist ein Agent im System vorhanden, der auf Tupel wartet (in), die von anderen Agenten in einen TS gestellt werden, und werden nahezu gleichzeitig mehrere passende Tupel in den TS gestellt, so können Tupel verloren gehen, d.h. der lesende Agent findet nicht alle in den TS gestellten Tupel.

2.4 Bewertung

Aus der Konzeption von Lime ergibt sich der erste grundlegende Kritikpunkt: Lime ist keine Implementierung von Linda, die alle Anforderungen der Spezifikation erfüllt. Im Wesentlichen ist Lime als Zusammenschluss einzelner Teil-TSs einzelner Agenten zu sehen. Bei diesen TSs wird nicht sehr weit vom zugrunde liegenden verteilten System abstrahiert, was vor allem bei den Locations deutlich wird. Somit ist Lime nicht ohne weiteres als Beispiel für einen Linda-TS zu verwenden. Ferner ist die fehlende Typsicherheit als weiterer Kritikpunkt anzusehen, da hier ebenfalls ein niedriges Abstraktionsniveau zu erkennen ist.

Da die Migration mittels µCode schwer realisierbar war[5] und die Migration über TSs ebenfalls mit einigen Problemen verbunden war, ist die Verwendung im Zusammenhang mit mobilen Agenten nicht ohne weiteres praktikabel.

Allgemein ist hervorzuheben, dass der Umfang von Lime sich auf grundlegende Bestandteile beschränkt und Anpassungen bzw. Erweiterungen den Entwicklern überlässt.

Besonders kritisch sind die beobachteten Systemausfälle zu bewerten. Hierbei handelt es sich um ein Ausschlusskriterium für den Einsatz in einer professionellen Umgebung, und selbst beim Einsatz im Rahmen eines Seminars oder ähnlichem ist, sofern ein komplexeres System entwickelt werden soll, von Lime abzuraten.

Der Einsatz von Lime ist dann anzuraten, wenn das zu entwickelnde System mit out und den blockierenden Operationen in und rd auskommt. Sind weitere Operationen erforderlich, so muss erreicht werden, dass den einzelnen Agenten bekannt ist, wohin Tupel geschrieben bzw. woher Tupel gelesen werden müssen. Auch ist hierbei die Unterscheidung zwischen der Spezifikation von Lime und Linda zu beachten. Ferner sollte das entwickelte System möglichst ohne Unsharing auskommen, um mögliche Systemausfälle zu vermeiden.

Als Vorteil von Lime ist die Abstraktion vom zugrunde liegenden Kommunikationsprotokoll hervorzuheben. Dieses bleibt für den Anwender – abgesehen von den Locations, die aber eher abstrakt sind – verborgen. Ferner ist der Einstieg in Lime sehr einfach, da der Anwender mit einem geringen Satz von Befehlen auskommt. Dies ist zum einen auf das Linda-Konzept, zum anderen aber auch auf die Implementierung von Lime zurückzuführen. In diesem Zusammenhang ist der weiter oben kritisierte Umfang von Lime als positiv anzusehen. Beim Einsatz von Lime ist auch sehr gut ersichtlich, dass das Grundkonzept von Linda vielfältige Möglichkeiten zur Simulation von Kommunikations- und Interaktionsprotokollen wie z.B. Message-Passing oder der Migration bietet. Zum Einsatz würden sich Übungen oder ähnliches eignen.

Da sich Lime in der Entwicklung befindet und auf der Website viele Anregungen und Pläne für zukünftige Versionen zu finden sind, erscheint es in jedem Fall sinnvoll die weitere

---

[5] Bei diesem wurde ein Konzept genannt ClassSpaces verwendet. Diese ClassSpaces sind Mengen von Klassen, die zur Migration vorgesehen sind, d. h. alle Klassen, welche von zu migrierenden Agenten verwendet werden. Diese ClassSpaces müssen den entsprechenden Servern, welche die Migration durchführen, explizit mitgeteilt werden.

Entwicklung im Auge zu behalten. Insbesondere geplante Erweiterungen, die über das Linda-Konzept hinausgehen – wie z.B. Transaktionen – wirken sehr interessant.

3 Das Casino-Konzept

3.1 Zielsetzung

Als Zielsetzung für das Seminar haben sich die Autoren ein System gesetzt, dass verschiedene Dienste, die im Zusammenhang mit agentenbasierter Software verwendet werden, mittels Lime simuliert und zur Verfügung stellt. Ferner sollte den Agenten die Möglichkeit gegeben werden, ihr Verhalten und ihre Fähigkeiten während der Laufzeit anzupassen. Hierzu wurden im Wesentlichen zwei Konzepte verfolgt. Zum einen wurden verschiedene Umgebungen mittels TS simuliert, in denen bestimmte Interaktions- bzw. Kommunikationsprotokolle gelten. In diesen Umgebungen werden die einzelnen Dienste realisiert. Zum anderen wurde die Interaktion innerhalb einer Umgebung nicht direkt durch einen Agenten sondern durch ein von ihm verwendetes so genanntes *Plugin* durchgeführt.

Umgesetzt wurde ein System auf dessen Basis, wie der Name bereits sagt, Spieler und Spielleiter zueinander finden und Spiele verschiedener Arten durchgeführt werden können. Alle Agenten in diesem System, bis auf die Spieler, wurden stationär umgesetzt, d.h. alle Administratoren und Spielleiter werden in jeweils einen LimeServer gestartet und verbleiben in diesem, bis sie das System verlassen oder das System heruntergefahren wird. Die Spieler demgegenüber suchen ausgeschriebene Spiele und migrieren zum LimeServer des jeweiligen Spielleiters.

3.2 Typsicherheit

Da Lime keine Typsicherheit für Tupel gewährleistet, wurden bei der Implementierung zwei unterschiedliche Verfahren umgesetzt.

Das erste Verfahren, welches im Zusammenhang mit den Umgebungen Lookup, Whiteboard und Migrationboard verwendet wurde, ist durch eine Kapselung der Tupel gekennzeichnet. Das jeweilige Plugin eines Agenten erzeugt hier die Tupel nicht selbst, sondern greift auf Objekte zurück, die als *Nachrichten* bezeichnet werden. Eine Nachricht kann entweder mit Hilfe eines gelesenen Tuples als Parameter oder mit den einzelnen Bestandteilen der Nachricht als Parametern initialisiert werden. Je nach Verwendungszweck kann vom Objekt ein Request- oder ein Result-Tupel erfragt werden. Wie bereits oben erläutert, ist diese

Unterscheidung erforderlich, da Lime für nicht gesetzte Felder bei Anfragen andere Inhalte erwartet als bei Tuplen, die in einen TS geschrieben werden. Mit Hilfe dieses Konzeptes lässt sich eine weitgehende Typsicherheit erreichen, wenn alle Agenten die gleichen kapselnden Klassen verwenden. Darüber hinaus können Änderungen, z.B. bei der Länge der Tupel oder bei der Position der einzelnen Einträge innerhalb eines Tuples, zentral vorgenommen werden. Ferner lassen sich durch dieses Konzept verschiedene Versionen von Interaktionsprotokollen realisieren.

Das zweite Verfahren wird im Rahmen des implementierten Spiels eingesetzt. Hier wird eine Klasse verwendet, welche die Templates der für das Spiel benötigten Tupel erzeugt. Diese können dann als Template für eine in- oder inp-Methode verwendet oder durch Füllen mit Werten in einen TS geschrieben werden. Diese Klasse verfügt dazu über public-Konstanten welche die Position einzelner Einträge in den Tupeln festlegen.

Diese beiden Konzepte können auch kombiniert werden, so dass das zweite Verfahren nicht Tupel, sondern die entsprechenden Nachrichten zurückgibt und so als Fabrik für Nachrichten eingesetzt werden kann.

3.3 Umgebungen

Im Folgenden werden die einzelnen Umgebungen kurz vorgestellt. Jede dieser Umgebungen, bis auf die Spielumgebung, ist systemweit einzigartig, d.h. für jede Umgebung existiert nur ein Administrator, und es existieren keine Umgebungen vom gleichen Typ parallel im System.

Lookup: Die Lookup-Umgebung dient als seine Art *Yellowpage* für das System. Jeder Agent, der in das System kommt meldet sich beim Lookup an und erhält eine eindeutige ID. Ferner werden im Lookup alle TSs mit ihrem Kontext, alle Spieltypen und alle laufenden Spiele eingetragen. Über den Lookup können Agenten folglich alle relevanten Informationen über das System erhalten. Der Lookup wird von einem *LookupAdministrator* verwaltet. Die Verwaltung beinhaltet zum einen die Initialisierung des Lookup-TSs und zum anderen die Vergabe der ID´s. Bei der Initialisierung stellt der Administrator ein *Versionstuple* in den TS. Dieses Tupel zeigt an, nach welchem Kommunikationsprotokoll im vorliegenden TS kommuniziert wird. Ferner stellt er ein *ID-Tupel* in den TS welches von einem neu hinzukommenden Agenten entnommen werden kann, um seine ID zu erhalten. Im weiteren Verlauf wartet der Administrator darauf, dass das ID-Tupel entnommen wird, um dieses zu ersetzen. Bei jedem Eintrag eines Agenten in den Lookup wird davon ausgegangen, dass der

jeweilige Agent alle Einträge in seinen eigenen Teil-TS schreibt. Der Administrator muss folglich nicht überprüfen, ob die einzelnen Einträge aktuell sind, da Einträge von das System verlassenden Agenten automatisch mit dem Unsharing entfernt werden. Wie oben bereits erwähnt, ist bei fast allen Operationen auf TSs die Angabe einer Location erforderlich. Daher wird bei jedem Eintrag eines Agenten die HostLocation und die AgentLocation des Agenten mit abgespeichert. Diese Umgebung muss, nachdem ein System-Leader gestartet wurde, als erste Umgebung gestartet werden, da alle nachfolgenden Agenten aus dem Lookup ihre ID beziehen und sich eintragen.

Pluginboard: Das Pluginboard dient den Spieler- und Spielleiteragenten als Bezugsquelle für Plugins. Der Administrator des Pluginboards ist, wie der Administrator des Lookups auch, für die Initialisierung des TSs für die Umgebung zuständig. Er registriert sich beim Lookup und setzt das Versionstuple. Anschließend schreibt er alle verfügbaren Plugins in den TS.

Whiteboard: Das Whiteboard dient als eine Art Treffpunkt für Spieler und Spielleiter. Hier können Spielleiter ihr Angebot an Spielen, für die noch nicht genug Spieler vorhanden sind, propagieren, und die Spieler können diese Informationen nutzen, um an Spielen teilzunehmen. Die Aufgaben des Administrators für diesen TS sind sehr beschränkt. Er stellt lediglich einen Versionstuple in den TS, meldet sich beim Lookup an und ermöglicht den Agenten seinen Teil-TS zur Kommunikation zu nutzen. Anders als die vorhergehenden TSs dient dieser nicht dem Speichern von Informationen, sondern als Simulation einer Art Message-Passing.

Migrationboard: Das Migrationboard dient dem Transfer von Agenten zwischen verschiedenen LimeServern. Mit ihm sind zwei Arten von Administrationsagenten verbunden. Zum einen ist ein *MigrationboardAdministrator* für die Verwaltung des TSs zuständig. Seine Aufgaben gleichen denen des Administrators des Whiteboards. Zum anderen Existiert für jeden LimeServer ein *Teleporter*, der auf Agenten wartet, die in den zugehörigen LimeServer migrieren wollen. Die Migration erfolgt über Tupel. Eine genaue Erläuterung der Migration erfolgt in Kapitel 3.7.

Spiel: Die für ein Spiel erforderlichen Nachrichten werden über einen für jedes Spiel angelegten TS ausgetauscht. Das Spiel wird vom sog. Spielleiter[6] organisiert. Dieser schreibt das Spiel im Whiteboard aus und wartet eine bestimmte Zeit auf Spieler, die teilnehmen

---

[6] Als Plugin für einen Agenten realisiert.

wollen. Sind für den Start des Spiels genügend Spieler vorhanden wird mit dem Spiel begonnen. Je nach Spiel ist der Spielleiter fähig während des Spiels neue Spieler ins Spiel aufzunehmen bis die maximal zulässige Anzahl an Spielern erreicht wurde. Ist das Spiel „voll“ wird der Eintrag für das Spiel vom Spielleiter aus dem Whiteboard entnommen. Der Spielleiter sorgt weiterhin dafür, dass die Spielregeln eingehalten werden. So kümmert er sich je nach implementiertem Spiel bspw. um die Einhaltung der Reihenfolge der Spieler, indem er die einzelnen Spieler zum Zug auffordert, die Züge auswertet, das Ergebnis der Züge allen Spielern mitteilt, Punkte an Spieler verteilt etc.

Die Spieler[7] versuchen im Whiteboard ein für sie passendes Spiel zu finden. Bei gefundenem Spiel versuchen sie sich bei diesem anzumelden. Ist die Anmeldung erfolgreich nehmen sie am Spiel teil, ist sie es nicht, versuchen sie ein neues Spiel zu finden. Dies geschieht ebenfalls nach Beendigung des Spiels.

Beim implementierten Spiel „Meier“ besteht der Spielleiter nur aus einer Komponente die sowohl die Nachrichten als auch die Auswertung dieser vornimmt. Der Spieler besteht aus einer Komponente, welche die Nachrichten erzeugt und einer Strategiekomponente, von der die Entscheidungen zur Reaktion auf diese Nachrichten gefällt werden[8].

3.4 Das Plugin-Konzept

Wie bereits erläutert, wurde für die Spieler- und Spielleiteragenten ein so genanntes *Plugin-Konzept* entwickelt. Ein Plugin stellt eine Kommunikationsschnittstelle für den Agenten mit einer Umgebung dar. Soll ein neuer Agent gestartet werden, so wird zunächst nur ein *CasinoAgent* gestartet. Dieser fungiert als eine Art Träger für die so genannte *Decision-Komponente*. Er verfügt lediglich über die Fähigkeit mit dem Pluginboard zu kommunizieren und hat darüber hinaus keine Aufgaben. Dem CasinoAgent wird nach dem Start mitgeteilt, welche Decision-Komponente er aus dem Pluginboard beziehen soll. Wurde die Decision-Komponente heruntergeladen und aktiviert, so übernimmt sie die Kontrolle. Der CasinoAgent hat nach dem Aufruf der run-Methode der Decision-Komponente eine passive Rolle und dient der Decision-Komponente als Schnittstelle zum Pluginboard. Die Decision-Komponente stellt die eigentliche *Intelligenz* des Agenten dar. In ihr werden die Entscheidungen getroffen und die Plugins verwaltet. Sobald die Decision-Komponente aktiviert wurde, fordert sie den

---

[7] Ebenfalls als Plugin realisiert

CasinoAgent auf weitere Plugins aus dem Pluginboard zu laden. Mit diesem initialen Set von Plugins nimmt sie ihre Arbeit auf.

Ist eine Interaktion mit einer Umgebung erforderlich, so wird das zuständige Plugin hiermit beauftragt. Durch das bereits beschriebene Problem, welches eine direkte Interaktion mit einem TS innerhalb einer extern aufgerufenen Methode verhindert, ist es erforderlich für jedes Plugin einen *CommAgent* und eine *OpQueue*[9] zu erstellen. Der CommAgent ist derjenige Bestandteil, der mit dem jeweiligen TS interagiert. Wird in einer Methode eine Interaktion mit einem TS erforderlich, so wird die dazu nötige Aktion in der OpQueue gespeichert. Die eigentliche Interaktion mit dem TS wird unter der Kontrolle der CommAgent-Thread ausgeführt. Diese wartet auf neue Einträge in der OpQueue und führt diese aus. Somit ist die kontrollierende Thread nicht mehr der Methodenaufrufer auf dem Plugin, sondern der CommAgent selbst. Die Interaktion mit dem TS erfolgt über die Java-Reflection-API.

Somit ergibt sich für einen Agenten folgender Aufbau:

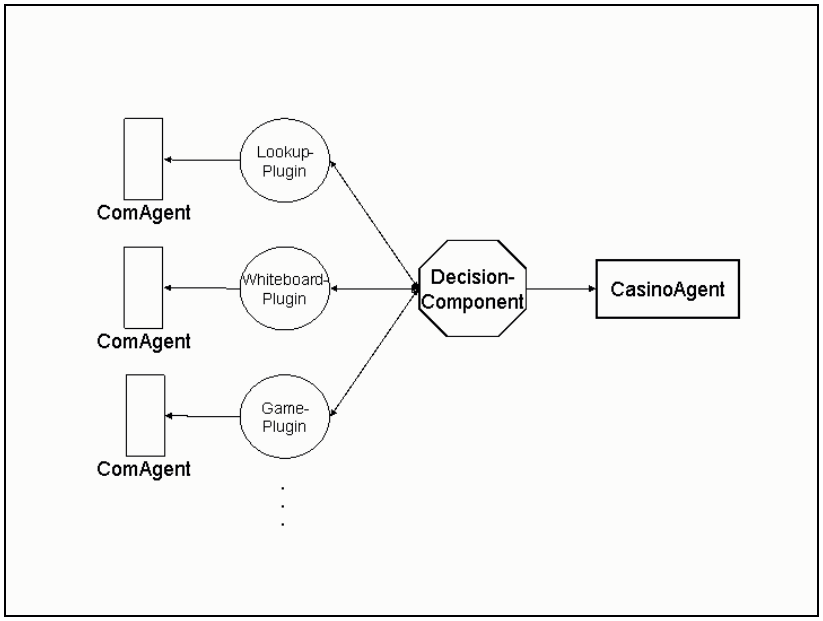

Abbildung 1: Aufbau eines Agenten

Die Decision-Komponente stellt zwar die zentrale Entscheidungsinstanz dar, es besteht aber auch die Möglichkeit, dass ein Plugin, zur Erfüllung einer ihm übertragenen Aufgabe, auf andere Plugins zurückgreifen muss. Wenn ein Plugin z.B. die Location eines Administrators

---

[8] Zu verbessern durch bessere Trennung von Kommunikation und Strategieentscheidung bzw. Spielsteuerung.

benötigt, so kann sie diese über den Lookup erfragen. Hierzu erfragt es das benötigte Plugin von der Decision-Komponente. Ist keine Instanz des benötigten Plugins vorhanden, so beschafft sich die Decision-Komponente diese aus dem Pluginboard-TS.

Das grundlegende Verhalten eines Agenten lässt sich somit nicht verändern, da die Decision-Komponente nicht ausgetauscht wird. Einem Agenten wird aber die Möglichkeit gegeben sich an veränderte Umweltbedingungen anzupassen. Inwiefern dies tatsächlich erfolgt, hängt entscheidend von der Implementierung der Decision-Komponente ab. Erhält sie bspw. den Auftrag sich nur an Spielen eines bestimmten Typs zu beteiligen, so ist die Anpassungsfähigkeit bzw. Erweiterbarkeit des Agenten zur Laufzeit stark eingeschränkt. Je mehr den Plugins Freiräume zur Aufgabenerfüllung eingeräumt werden, umso größer ist die zu erwartende Anpassungsfähigkeit des Agenten.

3.5 Spielleiter

Die Decision-Komponente für einen Spielleiter lädt, sobald sie ihre Arbeit aufgenommen hat, alle benötigten Plugins vom Pluginboard. Unter diesen befindet sich auch das Spielleiter-Plugin, welches als Letztes aktiviert wird.

Ist dieses aktiviert erzeugt eines einen TS für das Spiel und schreibt es im Whiteboard aus. Daraufhin wartet der Spielleiter auf Spieler, die am Spiel teilnehmen wollen. Sind genügend Spieler eingetroffen wird das Spiel gestartet. Der Spielleiter wartet eine gewisse Zeitspanne. Sind dann nicht genügend Spieler eingetroffen, werden die bereits angemeldeten Spieler wieder abgemeldet und das Spielleiter Plugin beendet sich und gibt den Grund der Beendigung an seine Decision-Komponente weiter. Treffen genügend Spieler ein wird dieses durchgeführt. Nun kommt es auf das einzelne Spiel an, ob während des Spiels noch weitere Spieler hinzukommen können bzw. bereits angemeldete Spieler sich abmelden können. Können keine Spieler mehr zum Spiel zugelassen werden, wird der Eintrag im Whiteboard entfernt.

Der Spielleiter fordert die Spieler in der den Spielregeln entsprechenden Reihenfolge zum Zug auf, nimmt den Zug entgegen wertet diesen aus, reagiert darauf, verteilt Punkte und informiert die anderen Spieler über die Vorgänge.

---

[9] Zu finden im Package `casino.agents.` bzw. `casino.misc.`

Nach Beendigung des Spiels werden die Spieler über das Ende des Spiels informiert und das Spielleiter-Plugin übergibt die Kontrolle wieder an seine Decision-Komponente mit entsprechendem Beendigungs-Status des Spiels.

3.6 Spieler

3.6.1 Computerspieler

Die Abläufe in der Decision-Komponente des Spielers sind analog zu denen im Spielleiter. Hier wird allerdings kein Spielleiter-Plugin, sondern ein Spieler-Plugin aktiviert. Dieses sucht ein Spiel über das Whiteboard. Nach erfolgreicher Suche meldet versucht es sich für dieses Spiel anzumelden. Dazu wartet es eine bestimmte Zeitspanne auf eine Bestätigung der Anmeldung. Wird die Anmeldung nicht bestätigt übergibt es die Kontrolle wieder an die Decision-Komponente.

Wird die Anmeldung akzeptiert nimmt der Spieler am Spiel teil. Er wartet auf die Aufforderung zum Zug vom Spielleiter und nimmt Informationen über den Spielverlauf entgegen, auf deren Basis er seine Entscheidungen trifft. Diese Entscheidungen werden vom Strategie-Plugin getroffen.

3.6.2 Menschliche Spieler

Für einen menschlichen Spieler wird das Strategie-Plugin durch ein GUI ersetzt, über das dann der Spieler die Züge bestimmen und den Spielverlauf verfolgen kann. Der Agent, welcher dieses GUI ‚trägt' sucht sich selbständig ein Spiel und meldet sich bei diesem an, so dass der menschliche Benutzer sich hierum nicht kümmern muss. Der Agent migriert allerdings nicht, sondern verbleibt in dem Lime-Server, auf dem er gestartet wurde. Beendet der menschliche Spieler das Spiel vorzeitig, übernimmt ein autonomes Strategie-Plugin seine Rolle.

3.7 Migration

In dem vorliegenden System migrieren die Computerspieler zu den jeweiligen LimeServern der Spielleiter, die ein Spiel ausrichten. Hierzu werden alle Plugins und die Decision-Komponente deaktiviert, indem ihre run-Methoden beendet werden. Die ComAgents der Plugins werden ebenfalls beendet. Hierzu löschen sie zunächst alle ausstehenden Tupel. Anschließend führen sie einen Unshare auf allen TSs durch. Einzig die TSs, die vom

CasinoAgent geshared wurden, bleiben erhalten. Stellt der CasinoAgent fest, dass alle Plugins inaktiv sind, so stellt er einen Tupel, der einen Migrationswunsch darstellt, in das Migrationboard. Hierin ist die Decision-Komponente mit allen zugehörigen Plugins enthalten. Der zuständige Teleporter entnimmt das Tupel aus dem TS und startet einen *MigratedCasinoAgent*. Dieser gleicht vom Aufbau und den aufrufbaren Operationen her dem CasinoAgent. Der einzige Unterschied besteht darin, dass die Decision-Komponente nicht neu aus dem Pluginboard bezogen wird, sondern darauf gewartet wird, dass die bei der Migration erhaltene Komponente übergeben wird. Diese wird auf dem neuen LimeServer zur Ausführung gebracht. Die Decision-Komponente reaktiviert daraufhin schrittweise alle Plugins. Bei der Aktivierung des Lookup-Plugins wird die bei der Erstellung des Agenten bezogene ID mit der neuen Location in die Yellowpages eingetragen. Da die run-Methode der Decision-Komponente bei jeder Migration neu gestartet wird, muss sich die Komponente den jeweiligen Zustand, in dem sie sich befindet, merken, d.h. zu Beginn ist sie bspw. in einem Zustand, in dem sie noch initialisiert werden muss und noch kein Spiel gefunden wurde und nach der Migration in einem Zustand, in dem sie sich, nach dem Reaktivieren der Plugins, bei einem Spiel anmelden muss. Es handelt sich folglich um zustandsbehaftete Agenten.

Bei dieser Vorgehensweise sind einige Besonderheiten zu beachten, die sich aus den Problemen bei der Anwendung von Lime ergeben. Theoretisch wäre das gerade beschriebene Verfahren unter Linda ohne weiteres durchführbar, da lediglich Lese- und Schreiboperationen stattfinden und sich Agenten bzw. Plugins bei TSs ab- und anmelden. Da aber das gleichzeitige Sharing und Unsharing regelmäßig zu abstürzen führten, musste die Migration synchronisiert werden. Darüber hinaus ergab sich das Problem, dass Agenten bei der Migration verloren gingen. Dies trat vor allem dann auf, wenn viele Agenten gleichzeitig zu demselben LimeServer migrieren wollten. Um dies zu verhindern wurde ein Verfahren eingesetzt, bei dem ein Synchronisationstoken mit zwei möglichen Zuständen zur Koordination der Migration eingesetzt wurde. Bei dem Token handelt es sich um ein Tupel, das als Eintrag den Zustand beinhaltet. Dieses Tupel liegt im Pluginboard-TS, da dieser TS als erstes shared und als letztes unshared wird. Im Zustand *free* hat ein Agent die Möglichkeit sich vom System abzumelden und sich in den Migrations-TS schreiben zu lassen. Hierzu entfernt er das Token aus dem TS. Nach Abschluss des Vorgangs stellt der CasinoAgent bzw. MigratedCasinoAgent das Tupel im Zustand *locked* zurück in den Pluginboard-TS. Ein Teleporter kann nur wenn das Token in diesem Zustand ist einen migrationswilligen Agenten aktivieren. Nach Abschluss der Aktivierung stellt er das Token wieder im Zustand free zurück. Das Token wird auch bei der Initialisierung von Agenten verwendet, da auch durch

diese die oben beschriebenen Probleme mit verursacht werden können. Allerdings wird das Token hier vor der Initialisierung im Zustand free entnommen und anschließend wieder im Zustand free zurückgestellt.

Da auch im Zusammenhang mit dem Spielleiter Probleme auftraten, wenn er ein Spiel beendet oder initialisiert, während andere Agenten migrieren, greift dieser ebenfalls auf das Token zu, damit die migrierenden Agenten nicht behindert werden.

Das beschriebene Verfahren bewirkt, dass im gesamten System immer nur ein Agent gleichzeitig migrieren kann bzw. initialisiert werden kann. Dies würde sich bei großen Systemen als Flaschenhals erweisen und wäre daher nicht praktikabel. Für das hier vorliegende System war es aber unvermeidlich, um eine Mindeststabilität zu erreichen.

3.8 Observer

Um einen Überblick über das System zu bekommen wurde ein GUI implementiert, welches die verschiedenen Umgebungen im System anzeigt. Diese Komponente nutzt zur Abfrage von vorhandenen Tupeln nicht-blockierende Leseoperationen. So lassen sich die Eintragungen für Spiele im Whiteboard, die vorhandenen Plugins im Pluginboard, die Einträge im Lookup etc. anzeigen. Weiterhin ermöglicht diese Komponente die Überwachung dynamischer Abläufe, wie z.B. das Hinzukommen von Spielen oder Plugins. Diese Funktion baut auf dem Konzept der Reactions auf, die den Observer über die stattgefundenen Tuplespace-Operationen informiert.

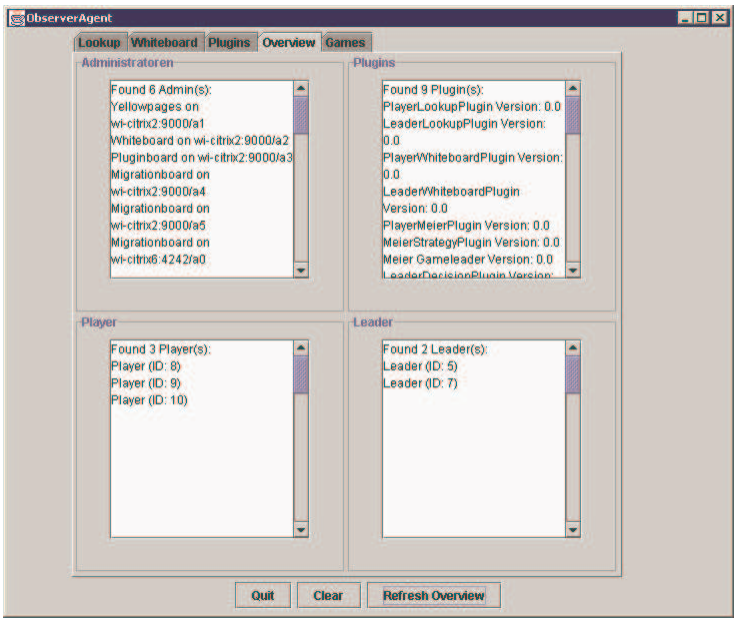

Abbildung 2: GUI des Observers

Der Observer kann zu jeder beliebigen Zeit gestartet werden. Aufgrund der Probleme im Zusammenhang mit Lime ist es allerdings ratsam den Observer möglichst nicht zu starten, während ein Agent migriert, oder ihn, ist er einmal gestartet, zu beenden. Hierbei kann es dazu kommen, dass das System steht oder einzelne Lime-Server crashen.

Über die Karteireiter der GUI lassen sich die verschiedenen Inhalte der TSs anzeigen. So sind unter ‚Lookup', ‚Whiteboard' und ‚Plugins' die Tupel bzw. Nachrichten zu sehen, die in den TSs für diese Umgebungen abgelegt sind. Unter dem Karteireiter ‚Overview' ist es möglich alle im System vorhandenen Agenten und alle Plugins aufzulisten, während der Karteireiter ‚Games' die Möglichkeit bietet alle momentan am Whiteboard ausgeschriebenene Spiele anzeigen zu lassen.

3.9 Das Spiel „Meier"

3.9.1 Spielregeln

Es wurde bis jetzt nur das Spiel Meier implementiert, bei dem es sich um ein Würfel-Spiel mit zwei Würfeln handelt. Das Ergebnis eines Wurfs ist eine zweistellige Zahl, welche sich aus den beiden Würfeln wie folgt zusammensetzt:

- Die hohe Augenzahl bildet die Zehnerstelle und die niedrige die Einerstelle des Ergebnisses. Zum Größenvergleich der Würfelergebnisse werden die entstehenden Beträge herangezogen.
- Werden gleiche Augenzahlen auf beiden Würfeln geworfen, ist dies ein Pasch, welcher höher bewertet wird als Würfe mit zwei verschiedenen Augenzahlen. Je höher die Augenzahl eines Pasches, umso höher wird er eingestuft.
- Ausnahme von diesen beiden Regeln bildet die „21", welche das höchste Würfel-Ergebnis darstellt und auch als „Meier" bezeichnet wird.

Die Spieler würfeln der Reihe nach und die Würfel und Spieler versuchen hierbei sich mit ihren erzielten Würfel-Ergebnissen zu übertreffen, wobei die Würfel allerdings verdeckt werden. Spieler A würfelt verdeckt, schaut sich sein Ergebnis an, gibt ein Ergebnis bekannt und reicht die Würfel verdeckt an seinen Nachfolger Spieler B weiter. B kann nun glauben, dass A das tatsächlich gewürfelte Ergebnis bekannt gegeben oder gelogen hat. Glaubt er A, würfelt er ebenfalls und gibt ein Ergebnis bekannt, das über dem von A verkündeten Ergebnis

liegen muss. Danach reicht er den Würfel verdeckt an seinen Nachfolger weiter, der die gleichen Möglichkeiten hat wie B.

Glaubt er A nicht, deckt er den Würfel auf. Hat A das richtige Ergebnis verkündet verliert B sonst verliert A. Der Verlierer kann mit unterschiedlichen Sanktionen (beispielsweise Minuspunkte für diesen) bestraft werden und/oder A, wenn B aufdeckte, obwohl er die Wahrheit sagte bzw. B, wenn er eine Lüge von A entdeckt, belohnt werden (beispielsweise mit Pluspunkten). In dieser Art setzt sich das Spiel fort bis ein Abbruchkriterium erreicht ist (beispielsweise 10 Runden gespielt oder Spieler A hat 20 Punkte etc.).

3.9.2 Das GUI des Spiels

3.9.2.1 Agenten

Um sich Informationen über den Spielverlauf zu schaffen, sind verschiedene GUIs verfügbar. Zum einen gibt es die GUI des Spielleiters, die nur auf dem Rechner angezeigt wird, auf dem der Spielleiter ausgeführt wird.

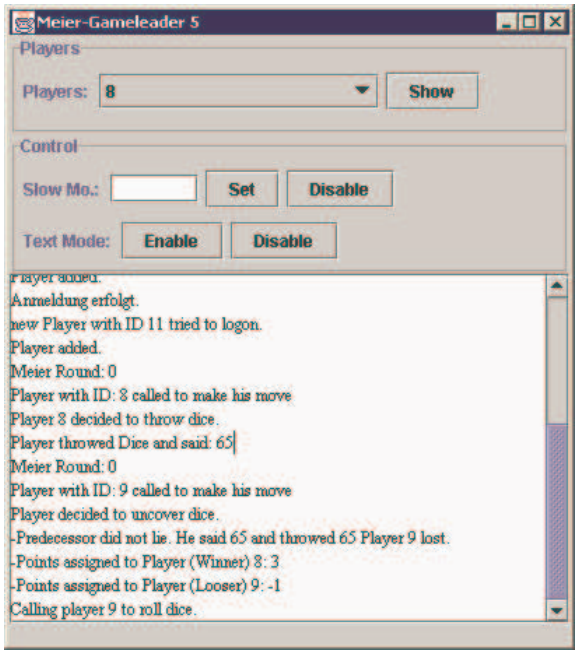

Abbildung 3: GUI des Spielleiters.

Das GUI besteht aus den Bereichen ‚Players', ‚Control' und einem Textfeld, in welchem der Spielverlauf in Textform angezeigt wird, um diesen zu verfolgen. Im Bereich ‚Control' in der Zeile ‚Text Mode' kann über den Button ‚Enable' der Spielleiter dazu veranlasst werden in dem Lime-Server, in dem er ausgeführt wird Textmeldungen über den Spielverlauf

anzuzeigen. Mit dem Button ‚Disable' werden die Textmeldungen wieder unterdrückt. Im Textfeld ‚Slow Mo.' lässt sich ein Zeitwert in Millisekunden eingeben, der dazu führt, dass der Spielleiter langsamer reagiert und der Spielverlauf leichter zu verfolgen ist. Über den nachstehenden Button ‚Disable' lässt sich dieser Wert wieder auf Null setzen.

Im Bereich ‚Players' werden in einer Auswahlliste die IDs der teilnehmenden Spieler angezeigt. Hier lässt sich dann ein Spieler auswählen und durch betätigen des Buttons ‚Show' ein GUI für diesen Spieler anzeigen.

Abbildung 4: GUI eines Computerspielers.

Dieses GUI zeigt den Status des Spielers an und verfügt ebenfalls über ein zum GUI des Spielleiters analog aufgebauten Bereich ‚Control'.

3.9.2.2 Menschlicher Spieler

Um am Spielgeschehen teilzunehmen, steht für den menschlichen Spieler ebenfalls ein GUI zur Verfügung, über das er dem Spiel entsprechende Eingaben tätigen kann. Über das Menü ‚Spiel' kann das Spiel verlassen werden, wenn der Spieler nicht gerade an der Reihe ist.

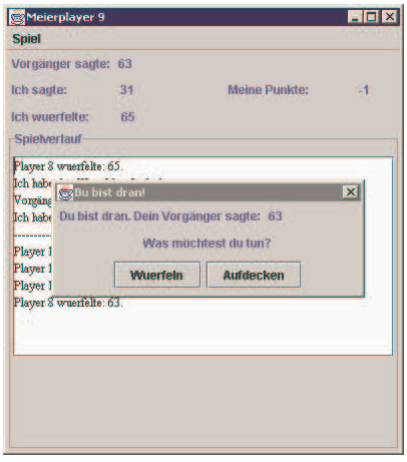

Abbildung 5: GUI für einen menschlichen Spieler.

Der Bereich unter dem Menü zeigt den Status an, während im Textfeld der Verlauf des Spiel ähnlich wie im GUI des Spielleiters angezeigt wird. Ist der Spieler am Zug wird er über einen Dialog aufgefordert entsprechende Züge zu machen.

4 Starten des Systems

Um das vorliegende System anzuwenden sind folgende Schritte nötig:

1. *Start des SystemLeader*: Bei dem ersten zu startenden Service handelt es sich um den SystemLeader. Dieser ist, wie bereits erwähnt, für die Koordination der LimeServer des Systems zuständig. Der SystemLeader ist durch den Befehl

   ```
   java lime.util.Launcher -port 9000 -load casino.agents.SystemLeader
   ```

   zu starten, wobei der Port frei wählbar ist. Der Launcher startet einen neuen LimeServer auf dem angegebenen Port und startet den SystemLeader in den neuen Server.

2. *Starten des Lookup*: Damit sich neu hinzukommende Agenten, Umgebungen und Spiele in die Yellowpages eintragen können ist direkt nach dem SystemLeader der Lookup zu starten. Dieser wird durch die Befehlszeile

   ```
   java lime.util.Launcher -quit -port 9000 -load casino.agents.environment.lookup.LookupAdministrator
   ```

   in den gleichen LimeServer wie der SystemLeader geladen. Die Option –quit weist den Launcher an keinen neuen LimeServer zu starten. Generell kann der Lookup auch in jeden beliebigen anderen LimeServer geladen werden. Hierzu ist lediglich die Option –quit zu entfernen und ein anderer Port zu wählen.

3. *Starten weiterer Umgebungen*: Nachdem der Lookup gestartet wurde, ist sind die anderen Umgebungen zu starten. Deren Start kann in beliegiger Reihenfolge stattfinden, da zwischen ihnen keine wechselseitigen Abhängigkeiten bestehen. Das Pluginboard wird durch die Befehlszeile

   ```
   java lime.util.Launcher -quit -port 9000 -load casino.agents.PluginBoard
   ```

   das Whiteboard durch die Befehlszeile

```
java lime.util.Launcher -quit -port 9000 -load casino.
agents.environment.whiteboard.WhiteboardAdministrator
```

und das Migrationboard durch die Befehlszeile

```
java lime.util.Launcher -quit -port 9000 -load
casino.migration. MigrationAdministrator
```

gestartet. Für alle Umgebungen gilt, dass sie in unterschiedliche LimeServer geladen werden können. Das laden aller Umgebungen in eine Umgebung erfolgte aus Performancegründen.

4. *Starten der Spielleiter*: Nachdem alle Umgebungen gestartet sind, können die Spielleiter gestartet werden. Ein Spielleiter wird durch die Befehlszeile

   ```
   java LeaderLauncher -port 4242
   ```

   gestartet, wenn für den Leader ein neuer LimeServer gestartet werden soll. Soll der Leader in einen bereits bestehenden LimeServer geladen werden, so sind die Befehlszeilen

   ```
   start java lime.util.Launcher -quit -port 9000 -load
   casino.migration.MigrationTeleporter
   ```

   und

   ```
   java lime.util.Launcher -quit -port 9000 -load casino.
   agents.LeaderAgent
   ```

   zur Ausführung zu bringen. Die zweite Befehlszeile sollte erst ausgeführt werden, wenn der MigrationTeleporter gestartet ist.

5. *Starten der Spieler*: Ein Computerspieler wird durch die Befehlezeile

   ```
   java PlayerLauncher -port 10040
   ```

   in einen neuen LimeServer bzw. mit

   ```
   java lime.util.Launcher -quit -port 9000 -load casino.
   agents.PlayerAgent
   ```

   in einen bereits bestehenden LimeServer geladen.

   Um als menschlicher Spieler an einem Spiel teilzunehmen, muss die Befehlszeile

```
java lime.util.Launcher -quit -port 9000 -load casino.
agents.human.meier.MeierHumanPlayerAgent
```

ausgeführt werden, bzw. ohne –quit mit einem anderen Port, um für den menschlichen Spieler einen eigenen LimeServer zu starten.

6. *Starten des Observer:* Der Oberserver zur Überwachung des Systems wird wie folgt gestartet:
   `java casino.agents.environment.observer.ObserverAgent`

5 Zusammenfassung und Ausblick

Das vorliegende System bietet eine Basis, auf der in einer verteilten Umgebung Spiele verschiedener Typen durchgeführt werden können. In diesem Kapitel wird zunächst beschrieben, wie ein neuer Spieltyp implementiert werden kann. Anschließend erfolgt eine Betrachtung aufgetretener Probleme, die bei einer Erweiterung zunächst zu beheben wären. Den Abschluss bildet eine Diskussion möglicher Erweiterungen.

Soll ein weiterer Spieltyp zur Verfügung gestellt werden, so sind im Wesentlichen zwei Plugins zu implementieren. Zunächst muss für den Spielleiter ein Plugin entwickelt werden, das es ihm ermöglicht ein Spiel des neuen Typs durchzuführen. Dieses Plugin muss vom Interface `casino.plugins.game.GameLeaderPlugin` erben. Die *run-Methode* des neu erstellten Plugins sollte folgendermaßen implementiert werden:

Initialiesierung Spiel-TS: Zunächst muss der TS initialisiert werden, auf dem das Spiel durchgeführt werden soll. Um zu verhindern, dass mehrere Spielleiter den gleichen Namen verwenden und so Überschneidungen auftreten können, bietet sich für die Benennung des TSs eine Kombination aus der ID des Spielleiters und dem Spieltypen an.

Spielersuche: Um Teilnehmer für das Spiel zu finden muss eine Nachricht vom Typ `casino.plugins.whiteboard.Version1_0.WBGameNachricht` in den Whiteboard-TS gestellt werden. Dies sollte unter Rückgriff auf das Whiteboard-Plugin erfolgen. Auf diese Nachricht reagierende Spieler werden sich über den Spiel-TS beim Spielleiter melden. Wichtig ist in diesem Zusammenhang, dass der Spielleiter die Nachricht entfernt, bevor er den Spielern mitteilt, dass das Spiel beendet ist. Tut er dies nicht, so kann es zu dem Effekt kommen, dass die Spieler versuchen erneut zu dem TS zu migrieren, dessen Spiel gerade beendet ist.

Durchführung des Spiels: Für die Durchführung des Spiels existieren keine Beschränkungen von Seiten des Systems. Der Spielleiter muss aber während der Durchführung des Spiels beobachten, ob sich neue Spieler anmelden wollen und diese ggf. annehmen oder abweisen bzw. den Eintrag des Spiels im Whiteboard löschen, wenn keine Spieler mehr teilnehmen können.

Mitteilung über Spielende: Ist das Spiel beendet, so muss der Spielleiter die Spieler hierüber informieren. Bei der Beendigung des Spielleiters müssen die unter Lime auftretenden Probleme berücksichtigt werden. Der Spielleiter darf nur dann seinen Spiel-TS als unshared deklarieren, wenn gerade kein Spieler migriert.

Die Implementierung des Spieler-Plugins erfolgt in ähnlicher Weise:

Suche des Spiels: Zunächst muss ein passendes Spiel mit Hilfe des Whiteboard-Plugins gesucht werden.

Migration zum Spiel-TS: Wurde ein passendes Spiel gefunden, so muss der Spieler zum LimeServer des Spielleiters migrieren.

Anmeldung und Spielen: Der Spieler sollte beim Versuch sich anmelden darauf gefasst sein keine Nachricht vom Spielleiter zu erhalten und aus diesem Grund nur eine bestimmte Zeit auf eine Anmeldung warten. Ist er angemeldet wird das Spiel nach den geltenden Spielregeln durchgeführt deren Einhaltung und Durchsetzung der Spielleiter übernimmt. Der Spieler muss also entsprechend auf die Nachrichten des Spielleiters reagieren können.

Nach Beendigung des Spiels hängt das weitere Verhalten der Spieler und des Spielleiters von den jeweiligen Decision-Komponenten[10] ab. Diese muss an die neuen Bedürfnisse angepasst werden.

Die in dem vorliegenden System verwendeten Komponenten sind ausschließlich für das Spiel Meier implementiert worden, d.h. sie laden nur Plugins für dieses Spiel aus dem Pluginboard-TS und suchen nicht nach Plugins für weitere Spiele. Denkbar sind zwei unterschiedliche Arten von Decision-Komponenten. Zum einen könnten für den Spieler und den Spielleiter universale Komponenten entwickelt werden, die nach allen möglichen Spielen suchen und diese spielen bzw. ausrichten. Zum anderen könnten für jedes Spiel spezielle Komponenten

---

[10] Abzuleiten von `casino.plugins.decision.DecisionPlugIn`.

entwickelt werden, diese würden analog zu den vorliegenden implementiert werden können. Generell ist abzuwägen ob ehr spezialisierte oder universale Agenten für das System wünschenswert sind.

Im vorliegenden System geht die Initiative zum Ausrichten eines Spiels von den Spielleitern aus. Denkbar wäre auch, dass die Spieler anfragen zur Ausrichtung eines Spiels in den Whiteboard-TS stellen. Dies wäre vor allem sinnvoll, wenn die Spielleiter als universale Agenten implementiert sind. In diesem Zusammenhang könnten sich auch Spieler untereinander zu Spielen verabreden. So könnten z.B. gemeinsame Strategien getestet werden oder Gruppenspiele mit nicht zufälligen Gruppen durchgeführt werden. Auch während eines Spiels könnte eine Kommunikation zwischen den einzelnen Spielern ermöglicht werden. Dies wäre vor allem für Gruppenspiele hilfreich um im Spielverlauf Absprachen zu ermöglichen, aber auch bei anderen Spielen könnte Spielern die Möglichkeit zur gegenseitigen Hilfe ermöglicht werden. Die vorliegenden Agenten verhalten sich weitgehend autonom. Durch eine Erweiterung um Kommunikation zwischen Spielern könnte ein Gruppen- bzw. Sozialverhalten beobachtet werden und es lassen sich über die Spiele hinaus differenziertere Strategien und Verhaltensweisen implementieren. Die Decision-Komponenten der Spieler im vorliegenden System haben keine Bewertungskriterien, nach denen sie sich verhalten. Sie suchen lediglich das nächste ausgerichtete Meier-Spiel und versuchen sich bei diesem anzumelden. Würden mehrere Spiele und die Kommunikation zwischen Spielern implementiert, so könnten verschiedene Kriterien bei der Spielbewertung eingeführt werden. Es könnten bspw. nach jedem Spiel individuelle Punkte und Punkte für die Spielergruppe vergeben werden. Je nach der von der Decision-Komponente vorgegebenen Präferenz könnte dann jeder Spieler seine Strategie individuell anpassen, je nachdem ob die Decision-Komponente den personlichen Erfolg oder den Gruppenerfolg höher bewertet. Denkbar wären auch Negativstrategien, bei denen es das Ziel eines Spielers ist einen oder mehrere andere Spieler zu schädigen.

Die Spielstrategien der Spieler sind im vorliegenden System statisch im Code vorgegeben. Die Spieler lernen nicht aus durchgeführten Spielen und es kommt zu keiner Anpassung der Strategien. Denkbar wäre z.B. der Einsatz neuronaler Netze oder regelbasierter Systeme zur Strategiebildung. Hierbei könnten die Spieler auch mehrere Strategien für das gleiche Spiel verwenden. So wäre bspw. eine Universalstrategie denkbar, neben denen verschiedene Individualstrategien für bestimmte Konstellationen von beteiligten Spielern eingesetzt werden können. Strategien könnten generell auf verschiedene Arten trainiert werden. Nach dem Training wäre beispielsweise eine Bereitstellung der Strategie im Pluginboard-TS möglich.

Die Spieler könnten diese Strategien unverändert übernehmen oder während der Spiele, an denen sie teilnehmen, anpassen. Ferner wäre ein so genanntes *Historyboard* denkbar, in das Berichte vergangener Spiele eingetragen werden. Anhand dieses Boards könnten Spieler ihre Strategien trainieren. Je nach eingesetztem Verfahren ist es möglich, dass Spieler sich darüber hinaus über ihre Strategien austauschen oder Strategien weitergeben. Würden lernende Strategien eingesetzt, so ist eine persistente Speicherung und Reaktivierung von Spielern als sehr sinnvoll anzusehen, damit die Strategien nicht bei jedem Systemneustart erneut angelernt werden müssten. Ferner ließen sich so gezielt verschiedene Konstellationen von Spielern untersuchen.

Abschließend sei herausgestellt, dass das vorliegende System eine Basis darstellt, auf der ein breites Spektrum von Spielen implementiert werden kann. Ferner ist durch das Plugin-Konzept und die Decision-Komponente die Möglichkeit gegeben, das System stark zu erweitern. Negativ wirken sich vor allem die Schwächen von Lime aus. So wird ein stabiler Einsatz des Systems stark behindert bzw. verhindert.

Literatur- und URL-verzeichnis

Literatur

O'Brien, P. D., Nicol, R. C. (1998), FIPA – towards a standard for software agents, BT Technol J Vol 16 No 3, S. 51-59, Juli 1998.

URLs

Lime: http://lime.sourceforge.net.

Lights: http://lights.sourceforge.net.

µCode: http://mucode.sourceforge.net.