MatchMaker – Flexible Synchronisation von Java-Applikationen

Marc Jansen, Niels Pinkwart, Frank Tewissen
{jansen,pinkwart}@informatik.uni-duisburg.de
F.Tewissen@gmx.de

Zusammenfassung
In diesem Artikel stellen wir ein System zur Synchronisierung beliebiger, in Java geschriebener, Anwendungen vor. Nach einer kurzen Diskussion der Vor- und Nachteile von replizierter Datenhaltung – wie sie in unserem System verwendet wird – werden wir am Beispiel einer komplexen Diskussions- und Modellierungsumgebung zeigen, wie man mit unserer Architektur Anwendungen partiell koppeln kann und somit flexible Kooperationsmöglichkeiten technisch unterstützen kann.

1 Einleitung
Im Bereich der intelligenten Lehr/Lernsysteme wird die Kooperation der Lernenden immer wichtiger. Um eine vernünftige Kooperation gewährleisten zu können, müssen die Anwendungen gekoppelt werden. Hierzu gibt es verschiedene Lösungsansätze, die im Wesentlichen auf zwei Modellen beruhen: zum einen kann man einfach die Daten, die zur Kopplung notwendig sind, auf einen zentralen Server legen und sie dort verwalten. Ein weiterer Ansatz beruht darauf, dass der Server nur für die Verteilung der Daten zuständig ist und die kompletten Daten an die Anwendungen weiterreicht (replizierte Architektur).
Zentrale Datenhaltung hat den großen Nachteil, das der einheitliche Zustand der Anwendungen nur solange gewährleistet ist, wie der Server im System erreichbar ist. Im Gegensatz hierzu ist bei replizierten Datenhaltungen üblicherweise gewährleistet, dass jede der verteilten Anwendungen zu jedem Zeitpunkt autark ist und als „stand-alone“ Applikation erhalten bleibt, auch wenn der Server ausfällt. Des weiteren ist es hier relativ einfach, nach einem Server-Ausfall wieder zu einem einheitlichen, gekoppelten Zustand aller Anwendungen zu kommen, da jede Applikation die Daten wirklich im Speicher hat. Es muss nur eine Applikation ihre Daten an den Server schicken, welcher sie dann wieder an die anderen Applikationen verteilt. Ein weiterer wichtiger Unterschied zwischen beiden Ansätzen liegt in der Entwicklung der Programme. Bei replizierten Anwendungen hat man die Möglichkeit, zuerst einmal die Anwendung als „stand-alone“ Applikation zu entwickeln und sich anschließend um die Kopplung zu kümmern. Damit ist durch dieses Modell ein hoher Grad an Wiederverwendung existierender Anwendungen und Source-Code möglich. Die Strategie zur Kopplung einer Applikation ist der zur Serialisierung (Abspeichern) der Daten sehr ähnlich und fällt daher in ähnlicher Form bei realen Anwendungen ohnehin an. In replizierten Architekturen ist weiterhin die Menge der Informationen, die übertragen werden müssen, durch geeignete Umsetzung (inkrementelle Operationen, also Beschränkung des Datentransfers auf diejenigen Daten, die sich tatsächlich geändert haben) auf ein Minimum reduzierbar. Ein wichtiger Nachteil replizierter Architekturen ist das Problem der Konsistenz, das bei redundanter Datenhaltung immer entsteht. Im Gegensatz zur zentralen Datenhaltung müssen konkurrierende Zugriffe explizit und mit erhöhtem Aufwand behandelt werden.
Im Folgenden möchten wir das Programm „MatchMaker“ vorstellen, das auf einer replizierten Architektur aufsetzt und zur Kopplung von Java-Applikationen verwendet werden kann. Die ersten Ansätze hierzu sind bereits zu finden bei Zhao und Hoppe (1994). Dort sind Überlegungen vorgestellt worden, wie man eine Applikation replizieren kann. Zum Beispiel wird schon in diesem Beitrag darauf eingegangen, wie man einfach über die Verteilung von Events die Daten in verschiedenen Anwendungen konsistent halten kann. Weiterentwicklungen des MatchMakers (Tewissen, Baloian, Hoppe und Reimberg, 2000) wurden im Rahmen des EU-Projekts NIMIS (Tewissen, Lingnau, Hoppe, Mannhaupt und Nischk, 2001) in Grundschulen eingesetzt.

2 Prinzip der Kopplung beim MatchMaker
Als einfaches Beispiel für die Anwendung des MatchMakers werden wir eine Beispiel-Applikation mit zwei eigenständigen Modulen synchronisieren. Das erste Modul verwendet eine komplexe Datenstruktur (einen Graphen), und findet z.B. zur Unterstützung von Diskussionsprozessen oder im Bereich des Wissensmanagements großen Anklang. Das zweite Modul – ein Chat als das "Hello World!" der verteilten Systeme – ist relativ einfach aufgebaut. Dabei wird die komplexe Datenstruktur die zur Verwaltung der

Kopplung nötigen Aufgaben selbst übernehmen; am Beispiel des Chats wird verdeutlicht, was zu einer Kopplung prinzipiell nötig ist.

Client-Sicht:
Die Kopplung setzt auf der von Sun bei Java mitgelieferten „Remote method invocation“ (RMI) Architektur auf. Aufgesetzt auf diese Architektur haben wir eine Objektmodell-Verwaltungsschicht, wobei wir als Modell eines Objektes nicht dessen graphische Repräsentation, sondern dessen interne Informationen (z.B. Userinterface-Parameter und weitere objektabhängige Variablen) bezeichnen. Dies hat den Vorteil, dass man wesentlich mehr als nur Benutzerschnittstellenobjekte koppeln kann (nämlich alle Objekte deren Modell aus serialisierbaren Objekten besteht). Weiterhin müssen die Anwendungen, die mit den Daten umgehen, nicht zwangsläufig ein- und dasselbe Modell auch durch ein- und dasselbe Benutzerschnittstellenobjekt repräsentieren, d.h. es bleibt der Anwendung (in Anlehnung an das Model-View-Controler Modell) überlassen, wie sie mit den Daten, die sie vom Server bekommt, umgeht und diese darstellt.
Die Anwendung hat sich lediglich darum zu kümmern, dass sie im gekoppelten Modus
a) alle geänderten Objekte, in Form der jeweiligen Modelle, in den Server schreibt
b) die geänderten Modelle, die sie vom Server bekommt, lokal umgesetzt werden

Damit die Anwendung mit Sicherheit die im Server anfallenden Änderungen erhält, muss sie sich, gemäß dem Java Event Model, als „Listener“ beim Server registrieren. Dazu müssen die Applikationen ein Interface implementieren (SyncListener), das die folgenden abstrakten Methoden beinhaltet:

- objectCreated (wird aufgerufen, wenn ein Objekt im Server angelegt wird)
- objectChanged (wird aufgerufen, wenn ein Objekt im Server verändert wird)
- objectDeleted (wird aufgerufen, wenn ein Objekt im Server gelöscht wird)
- actionExecuted (wird aufgerufen, wenn auf einem Objekt im Server eine Aktion ausgeführt wird)

Server-Sicht:
Dem Server fällt nun zum einen die Aufgabe zu, seine „SyncListener“, also die an den Events interessierten Anwendungen, zu verwalten, zum anderen muss er die Objekte, die über ihn verteilt werden, in geeigneter Weise abspeichern. Die Speicherung im Server findet beim MatchMaker in Form eines Baumes statt. Das hat den Vorteil, dass ein Client sich nicht notwendigerweise für alle Objekte im Server als Listener registrieren muss, sondern dies für spezielle Teilbäume tun kann. Damit werden an ihn nur die Events gesendet, die in diesem Teilbaum auflaufen. Haben wir zum Beispiel eine Applikation, die als Arbeitsbereich einen Graphen hat und als Kommunikationsbereich einen Chat nutzt, so sind für den „Chat-Teil“ der Applikation die Events über Graph-Änderungen nicht von Interesse. In diesem Fall würde man als einen Teilbaum den Graphen und anderen Teilbaum den Chat (bzw. dessen Modell) im Server speichern.

Des weiteren ist es möglich, in einem MatchMaker-Server verschiedene Gruppen von Anwendern oder Applikationen in sogenannten Sessions zu verwalten. Dazu muss die Applikation dem Server nur mitteilen, an welcher Session sie teilnehmen will bzw. welche Session sie eröffnen möchte . Dies wird in einfacher Weise von dem Server unterstützt, wie folgender Codeausschnitt zeigt:

```
public void createSession()
{
  [...]
    // Hier legen wir eine neue Instanz von unserem MatchMaker-Client an
    MmClient mm=new MmClient(serverHostName);
    // Die Session bekommt einen Namen
    String name="ourSession";

    // Anlegen der Session
    mm.createSession(new SyncLabel(name));
    // Die internen Namen werden gesetzt
    SyncLabel rootLabel=mm.newChildLabel(SyncTree.ROOT);
    SyncLabel graphLabel=mm.newChildLabel(SyncTree.ROOT);
    SyncLabel chatLabel=mm.newChildLabel(SyncTree.ROOT);
    // Hier wird der Baum, der später in den Server geschrieben wird, konstruiert
    SyncTree root=new SyncTree(SyncTree.ROOT,rootLabel);
   // Da der Graph sich als komplexe Datenstruktur selbst verwalten kann, rufen wir hier
   // nur eine Methode im Graphen auf, die alles weitere veranlasst
    SyncTree graphTree=graph.init(mm,graphLabel);
    SyncTree chatTree=new SyncTree(chatLabel,"");
    root.addChild(graphTree);
    root.addChild(chatTree);
    // Der Baum wird in den Server geschrieben
    mm.writeSyncTree(root);
    // Hier teilen wir dem Server mit das wir an den Events, die den Chat betreffen,
    // interessiert sind
    mm.addSyncListener(chatLabel,this);
    // Für den Graphen brauchen wir keinen SyncListener, da er sich selbst verwaltet
```

```
  [...]
}

public void joinSession()
{
  SyncLabel graphLabel;

 [...]
    // Instanziieren des MatchMaker Clients
    mm=new MmClient(serverHostName);

    // Angabe der Session, an der wir teilnehmen wollen
    SyncLabel session="ourSession";
    // Join zur Session
    mm.joinSession(session);
    // Lesen des Baums aus dem Server
    SyncTree tree=mm.readSyncTree(SyncTree.ROOT);
    // Lesen der einzelnen Unterbäume
    tree=tree.childAt(0);
    ...
    tree=tree.childAt(1);
    ...
    // Abschließend registrieren wir uns wieder als Listener für den Chat
    mm.addSyncListener(chatLabel,this);
  }
  [...]
}
```

3 Vergleich zu anderen Architekturen für replizierte Anwendungen

In diesem Abschnitt werden wir unsere Architektur zum Synchronisieren von Applikationen mit anderen Architekturen vergleichen. Wir legen bei dem Vergleich besonderen Wert auf Geschwindigkeit, interne Datenstrukturen sowie die Möglichkeit, Applikationen später als andere in die Sessions aufzunehmen (Bedienung von „late-comers“).

JavaSpaces

Sun bietet mit „JavaSpaces“ eine Umgebung an, die prinzipiell dem skizzierten Ansatz von MatchMaker sehr ähnelt. Allerdings ist die interne Speicherung bei JavaSpaces nicht in Form eines Baumes realisiert, was Vor- und Nachteile hat. Der wohl offensichtlichste Vorteil liegt darin, dass der Verwaltungsaufwand im Server wesentlich geringer ist. Dafür hat man größere Probleme, wenn man Daten strukturiert im Server ablegen möchte. Den gesamten Verwaltungsaufwand für diese Struktur muss man selber tragen. Das führt in der Realität zu sehr großen Geschwindigkeitseinbußen. Des weiteren haben JavaSpaces die Nachteile, dass sich Applikation nicht als Listener für einen Teil einer Anwendung registrieren können und dass ein Server nicht mehr als eine Session verwalten kann. Der größte Schwachpunkt von JavaSpaces liegt unserer Meinung nach darin, dass keine Transaktionssicherheit besteht – Applikationen werden nicht benachrichtigt, ob ein Objekt erfolgreich in den Space geschrieben wird. Dies führt in der Praxis a) zu großen Performance-Problemen und b) zu "verschlafenen" Events, d.h. zu inakzeptabel langen Zeiten von inkonsistenten Zuständen in der Gruppe der verteilten Anwendungen.

JavaSpaces unterstützen „late-comers“ generisch, denn da die Synchronisationsobjekte im Space verbleiben, muss eine nachträglich zu synchronisierende Applikation nur alle sie interessierenden Objekte aus dem Space lesen wenn sie die Session betritt. Zusätzlich bieten JavaSpaces die Möglichkeit, durch einen einfachen "Schalter" den Zustand eines Spaces persistent abzulegen und damit für eventuelle Server-Ausfälle oder spätere Wiederaufnahmen der Sitzung vorzuhalten.

Java Shared Data Toolkit (JSDT)

Eine weitere Alternative zu MatchMaker oder JavaSpaces kommt auch von Sun und heißt „Java Shared Data Toolkit“ (JSDT). Auf den Webseiten von Sun findet man folgende kurze Beschreibung der Einsatzgebiete von JSDT: „Enterprise developers can use the Java Shared Data Toolkit software to create network-centric applications, such as shared whiteboards or chat environments. It can also be used for remote presentations, shared simulations, and to easily distribute data for enhanced group workflow.“ Dies liegt – wie auch JavaSpaces – relativ nah an den Einsatzgebieten des MatchMakers. Allerdings hat JSDT den Nachteil, dass jegliche Kommunikation an Kanäle, sogenannte Channel, gebunden ist. Alle Informationen eines Kanals gehen an die für den jeweiligen Channel registrierten Applikationen, die dann ihrerseits entscheiden müssen welche der Informationen für sie interessant sind. Dies erfordert einen hohen organisatorischen Aufwand, weiterhin ist die Abbildung komplexer Strukturen auf die Kanäle oft nicht ohne Weiteres möglich.

Das Problem der „late-comers“ ist bei JSDT sehr viel komplexer als bei JavaSpaces. Da die Kommunikation in Kanälen abläuft, müsste einer später kommenden Applikation die gesamte bisherige Kommunikation eingespielt werden. Insbesondere müsste also der Server eine Historie der Kommunikation mitführen und verwalten. JSDT ist auf Performance ausgerichtet und eignet sich sehr gut für verteilte Echtzeit-Synchronisation von Streaming-Data. Ein Audio-Chat lässt sich beispielsweise sehr elegant mit JSDT umzusetzen.

Frühere Versionen von MatchMaker
Die erste Implementation von MatchMaker wurde von Frank Tewissen an der GMD in Darmstadt in Zusammenarbeit mit Prof. Hoppe in C++ realisiert. Diese Version wurde auch zum Koppeln eines CardBoards, das ebenfalls in C++ geschrieben war, verwendet. Die neueren Versionen sind durchweg in Java implementiert und basieren ausschließlich auf RMI. Die ersten Versionen von MatchMaker synchronisierten einfache Events, der Benutzerschnittstellenobjekte (z.B. Aktivieren einer Checkbox). Nachteile, die durch dieses Konzept entstanden, sind in der neuesten Version (MatchMaker TNG) beseitigt. In dieser werden nicht mehr die Events synchronisiert, sondern Repräsentationen der betreffenden, zu synchronisierenden Objekte (Modelle). Dadurch wurde die Klasse der koppelbaren Objekte wesentlich vergrößert.
Da in den älteren MatchMaker-Versionen, ähnlich zur Vorgehensweise bei JSDT, die Kopplung durch Übertragung der Events stattfand, ist der nachträgliche Zugang zu Gruppensitzungen schwierig. In einem solchen Szenario muss der Server entweder eine Historie der Events mitführen oder die Möglichkeit haben, jederzeit ein Abbild der gekoppelten Applikationen zu erstellen. Dies erfordert aber einen weiteren organisatorischen Overhead, den wir in der neuen MatchMaker Version vermieden haben.
Machbarkeitsstudien sowohl mit JavaSpaces als auch mit JSDT führten zu der jetzt entstandenen Architektur.

4 Protokollierungsfunktionen des MatchMakers

Ein wichtiges und oft nachgefragtes Feature von Groupware ist die Möglichkeit der automatischen Protokollierung der Benutzerinteraktion, z.B. zu Zwecken der Evaluation. MatchMaker unterstützt dies generisch und erstellt parallel zur Verwendung der gekoppelten Applikationen eine Protokollierungsdatei. Diese liegt im XML-Format vor und kann so auf einfache Weise analysiert und weiterverarbeitet werden. Hier sehen wir eine Beispieldatei von einer diskussionsunterstützenden Anwendung, die mittels MatchMaker gekoppelt wurde:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE EasyDiscussingLog SYSTEM 'file:/Y:/dfg/dfg/udui/main/dtd/EasyDiscussingLog.dtd'>

  <EasyDiscussingLog>
    <TextNodeAdded>
      <id>026fabac1a95a213:f73c1:e79ce93e8b:-7fd3</id>
      <label>//2/5/8</label>
      <user>niels</user>
      <date>Tue Jul 10 15:00:12 CEST 2001</date>
    </TextNodeAdded>
    <AnnotationNodeAdded>
      <type>1</type>
      <id>026fabac1a95a213:f73c1:e79ce93e8b:-7fc7</id>
      <label>//2/5/10</label>
      <user>niels</user>
      <date>Tue Jul 10 15:00:17 CEST 2001</date>
    </AnnotationNodeAdded>
    <Chat>
      <label>//3</label>
      <text>niels: hallo marc (!)</text>
      <date>Tue Jul 10 15:00:22 CEST 2001</date>
      <user>niels</user>
    </Chat>
    <Chat>
      <label>//3</label>
      <text>marc: hallo niels (!)</text>
      <date>Tue Jul 10 15:00:34 CEST 2001</date>
      <user>marc</user>
    </Chat>
    <EdgeAdded>
      <id1>026fabac1a95a213:f73c1:e79ce93e8b:-7fd3</id1>
      <id2>026fabac1a95a213:f73c1:e79ce93e8b:-7fc7</id2>
      <user>marc</user>
      <date>Tue Jul 10 15:00:40 CEST 2001</date>
    </EdgeAdded>
  </EasyDiscussingLog>
```

5 Das JCardBoard: Flexible Kooperationsunterstützung

Nachdem in den vorherigen Absätzen dieses Artikels die technischen Grundlagen der Synchronisationsmechanismen des MatchMakers beschrieben und diskutiert worden sind, möchten wir abschließend ein Beispielsystem vorstellen, welches die von MatchMaker gebotene Flexibilität ausnutzt, um Benutzern unterschiedlichste Arten der computergestützten Kooperation zu ermöglichen.
Dieses System, das JCardBoard, stellt den Benutzern, ähnlich wie etwa in Belvedere (Suthers, Weiner, Connelly und Paolucci, 1995) oder gIBIS (Conklin, und Begemann, 1987) , visuelle Sprachen zur Verfügung, mit denen

z.B. Begriffsnetze konstruiert oder Diskussionsabläufe strukturiert darstellt werden können. Als zentrale Erweiterung zu den genannten vergleichbaren Systemen bietet es die Möglichkeit, domänenabhängige Inhalte sowie Semantik in die allgemeine Diskussionsumgebung flexibel zu integrieren. (Pinkwart, Hoppe und Gaßner, 2001) Existierende Beispiele für die Nutzung des JCardBoards (und somit für die Möglichkeiten der Integration von domänenabhängigen Elementen) sind Messwertanalyse und Datenmodellierung in Naturwissenschaften, die Simulation von Petri-Netzen oder System Dynamics Modellen sowie die Nutzung verschiedener multimedialer Inhalte zur Vermittlung der jüdischen Kultur, Schrift und Sprache.
Die genannte Bandbreite von Anwendungen ist möglich aufgrund folgender Systemstruktur:
Die Integration von domänenabhängigen Inhalten ist über extern definierbare „Paletten“ realisiert, die entweder beim Start des Programms aus einer XML-Konfigurationsdatei oder dynamisch durch den Benutzer geladen werden können. Paletten enthalten die für die vorgesehene Nutzung des JCardBoards notwendigen Elemente, z.B. die in Abbildung 1 dargestellten typisierten Diskussionsbeiträge , Stellen und Transitionen für Petri-Netze oder Möglichkeiten der Parametrisierung von Handschrifteingaben (Strichdicke, -farbe etc.).

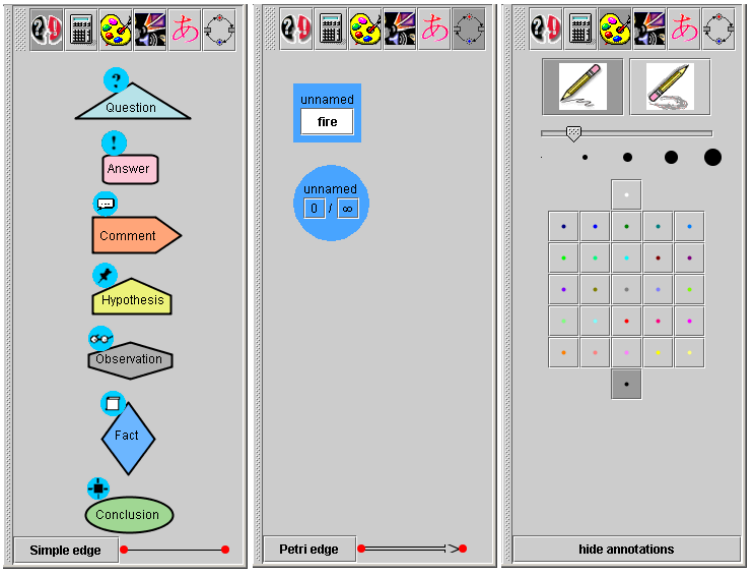

Abbildung 1: Beispiele für Paletten

Während die Paletten den statischen Teil des JCardBoards bilden und die Vorlagen für die Nutzung enthalten, finden die eigentliche Verwendung, z.B. die Diskussion oder die Messwertanalyse, in Arbeitsbereichen statt, von denen das JCardBoard mehrere, dargestellt durch verschiedene Fenster, verwalten kann. Dies bietet die Möglichkeit, private und synchronisierte Aufgaben zeitgleich zu bearbeiten. Jeder dieser Arbeitsbereiche ist in eine Anzahl von (transparenten) Ebenen unterteilt, die etwa Bilder, Handschriftstriche oder auch Diskussionsbeiträge enthalten können – allgemein genau die Elemente, die von den Paletten geboten werden. Wie die Arbeitsbereiche selbst, können die Ebenen privat oder synchronisiert sein, was erneut eine erhebliche Erweiterung der Kooperationsmöglichkeiten mit sich bringt, wie in Abbildung 2 zu sehen ist. Hier sind zwei teilweise synchronisierte Anwendungen zu sehen, bei denen ein Arbeitsbereich (links) vollständig und ein zweiter (rechts) nur auf einer Ebene synchronisiert ist. Einer der Benutzer hat weiterhin einen privaten Arbeitsbereich (Mitte)

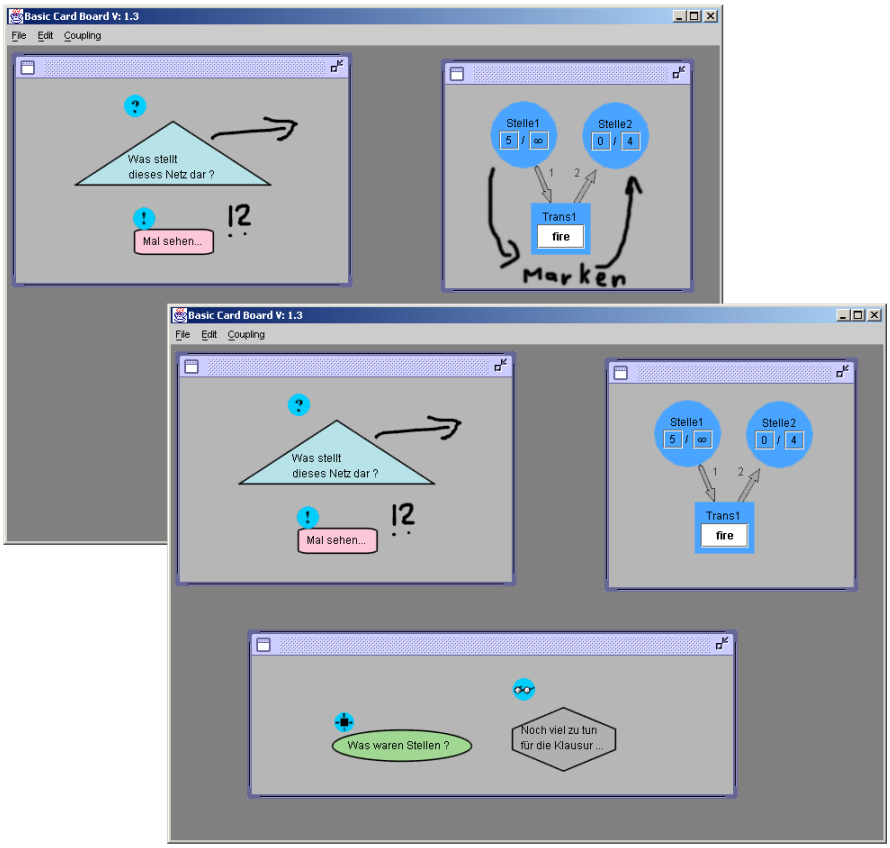

Abbildung 2: Teilweise gekoppelte Arbeitsbereiche

Wie bereits angedeutet, verwendet das JCardBoard den MatchMaker als zentrales Synchronisationsmittel, wodurch die in obiger Abbildung verdeutlichte Flexibilität in der Definition der synchronisierten Elemente zwischen verschiedenen JCardBoard-Instanzen relativ leicht zu erzielen ist. Die zugrunde liegenden Prinzipien der Kopplung sind:

1) Der im MatchMaker erzeugte Synchronisationsbaum entspricht der logischen Hierarchie (Anwendung-Arbeitsbereich-Ebene) innerhalb des JCardBoards
2) Alle Arbeitsbereiche sind im Synchronisationsbaum enthalten, wobei *privat* genutzte Elemente genau eine Listener-Anwendung, *kooperativ* genutzte Elemente hingegen mehrere, besitzen

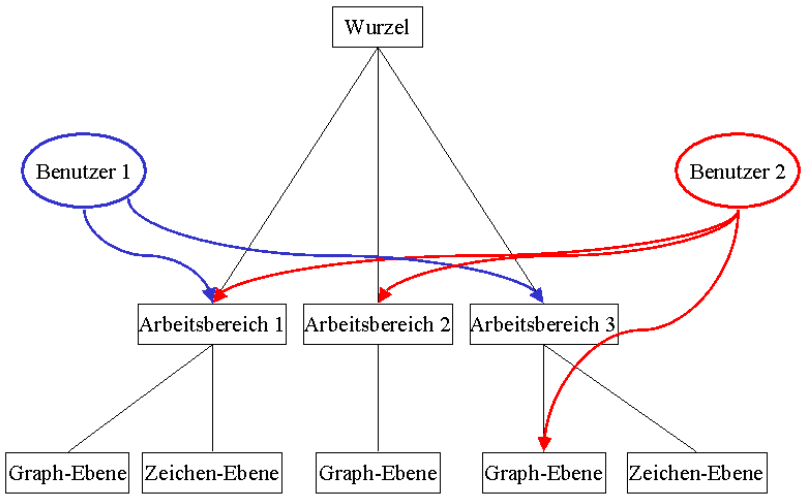

Abbildung 3: zu Abbildung 2 gehörender Synchronisationsbaum

6 Ausblick

Bei der Entwicklung von zukünftigen MatchMaker-Versionen werden im Wesentlichen drei Ziele eine Rolle spielen: *Erreichung von Persistenz*, *Weiterentwicklung der Protokollierungsmöglichkeiten* sowie *Erleichterung einer Datenbankanbindung*.

Diese Ziele lassen sich auf normierte und relativ leichte Weise durch den weiteren Ausbau der XML-Unterstützung innerhalb des MatchMakers erreichen. Wir werden wir die Möglichkeit untersuchen, die Modelle der zu koppelnden Objekte in einer XML Repräsentation zu versenden. Damit könnte eine leichte Anbindung an existierende Datenbanken realisieren, sei es zur dauerhaften Speicherung der Gruppenergebnisse oder zum Protokollieren von Aktionen.

Unter persistenter Speicherung verstehen wir die Möglichkeit, dass der Server zu jeder Zeit seinen kompletten Inhalt speichern kann, um ihn bei eventuellen Abstürzen oder Neustarts wieder herzustellen. Die Speicherung des Inhalts sollte im XML Format geschehen, was begünstigt würde durch die Tatsache, dass sich die Modelle der Objekte in XML leicht repräsentieren lassen. Weiterhin speichern Anwendungen ihre Daten üblicherweise ohnehin ab, und wenn sie das in einem XML-Format tun, ist der Schritt über die Kopplung hin zur persistenten Speicherung offensichtlich klein. Unterstützend wird sich auswirken, dass in der neuen Java-Version genau eine solche XML-Serialisierung implementiert sein wird.

Des weiteren werden wir die XML-Protokollierungsmöglichkeiten des Servers erweitern mit dem Ziel, die Inhalte der Logdatei extern, also durch die gekoppelte Anwendung, definierbar zu machen (unter Beibehaltung der generischen Protokollierung als „default“-Fall).

7 Danksagungen

Teile der in diesem Artikel vorgestellten Arbeiten wurden im Rahmen des DFG-Schwerpunktprogramms „Netzbasierte Wissenskommunikation in Gruppen“ sowie des EU-Projekts „DiViLab“ erstellt und gefördert.

8 Literatur

Conklin & Begemann (1987): „gIBIS: A hypertext tool for team design deliberation” In Proceedings of Hypertext’87 (pp. 247-251). Chapel Hill, NC (USA).

DiViLab: Distributed Virtual Laboratory. EU-gefördertes Projekt im fünften IST-Rahmenprogramm, Projektnummer 12017. http://www.divilab.org

Freeman, Hupfer & Arnold (1999): „JavaSpaces Principles, Patterns and Practice“, Addison Wesley

JavaSpaces: http://java.sun.com/products/javaspaces/index.html

JSDT: http://java.sun.com/products/java-media/jsdt/index.html

Netzbasierte Wissenskommunikation in Gruppen: Projektnummer HO 2312/1-1 im DFG-Schwerpunktprogramm

NIMIS: Networked Interactive Media in Schools. EU-gefördertes Projekt im vierten IST-Rahmenprogramm, Projektnummer 29301. http://collide.informatik.uni-duisburg.de/Projects/nimis

Pinkwart, Hoppe & Gaßner (2001): "Integration of Domain-specific Elements into Visual Language Based Collaborative Environments“. In Proceedings of 6th International Workshop on Groupware, CRIWG 2001 Darmstadt, Germany, IEEE CS Press.

Suthers, Weiner, Connelly & Paolucci (1995): „Belvedere: Engaging students in critical discussion of science and public policy issues” In Greer, J. (ed.), Proceedings of the 9th World Conference on Artificial Intelligence in Education (pp. 266-273). Washington DC (USA).

Tewissen, Baloian, Hoppe & Reimberg (2000): „MatchMaker - Synchronising Objects in Replicated Software-Architectures“. In Proceedings of 6th International Workshop on Groupware, CRIWG 2000 Madeira, Portugal, IEEE CS Press.

Zhao & Hoppe (1994): „Supporting Flexible Communication in Heterogeneous Multi-User Environments“ In: Proceedings of the 14th international conference on distributed computing systems. IEEE Computer Society Press.

Zumbach, Muehlenbrock, Jansen, Reimann & Hoppe (2001): „Multi-Dimensional Tracking in Virtual Learning Teams. An Exploratory Study”, eingereicht zur CSCL 2002