MatchMaker - A Framework to Support Collaborative Java Applications

Marc Jansen
Gerhard Mercator University of Duisburg
jansen@collide.info

Abstract. The work presented in this paper deals with the MatchMaker framework that was developed to easily extend stand-alone Java applications to collaborative ones. This is achieved by a combination of a replicated architecture with a centralized server and with respect to well-known design patterns like the Model-View-Controller (MVC) architecture. Furthermore several additional functions have been implemented such as a replay function that enables the user to replay sessions after they took place. This replay function is based on an advanced logging mechanism that also supports an undo/redo framework that enables us to undo/redo actions in collaborative sessions. After presenting the basic functions of the framework, an outlook to the future work concerning MatchMaker is given.

Introduction

Since over the last few years the influence of networked computers plays a more and more important role in the area of shared workspace environments the need to easily extend stand-alone applications to collaborative ones gets even more important. To support this work as much as possible by the mean of enabling developers to do this extension in an intuitiv and straight-forward way it is necessary to have collaboration frameworks.
One major element of the MatchMaker framework is that it is based on well-known design patterns like the Model-View-Controller (MVC) architecture. MatchMaker supports collaboration by combining two architectures for collaborative systems: On the one hand the framework has got a centralized server and on the other hand it is based on a replicated architecture. Combining these two solutions we got the advantages of both with removing the disadvantages: For example one problem of a centralized server is that given this server is no longer reachable it is not possible for the clients to work any longer. Using a replicated architecture enables us to continue working stand-alone even if the server is no longer reachable. After reconnecting the server to the network it is only necessary that one client sends his data to the server so that every other client can get the data from the server again. On the other hand it is much easier to implement persistency if all the data is saved in one central instance.
Furthermore, MatchMaker allows partial coupling of applications by using data structures that reflect the applications internal structures. This paper also presents a replay function and an undo/redo framework, both based on an advanced logging mechanism.

1. MatchMaker

Since MatchMaker combines two architectures for distributed systems, on the one hand a centralized server architecture and on the other hand a replicated architecture, each MatchMaker clients has to register itself as a listener at a MatchMaker server instance. Internally in the MatchMaker server the data is organized as a synchronization tree that

reflects the internal data structure of the application. Figure 1 shows an example of such a synchronization tree.

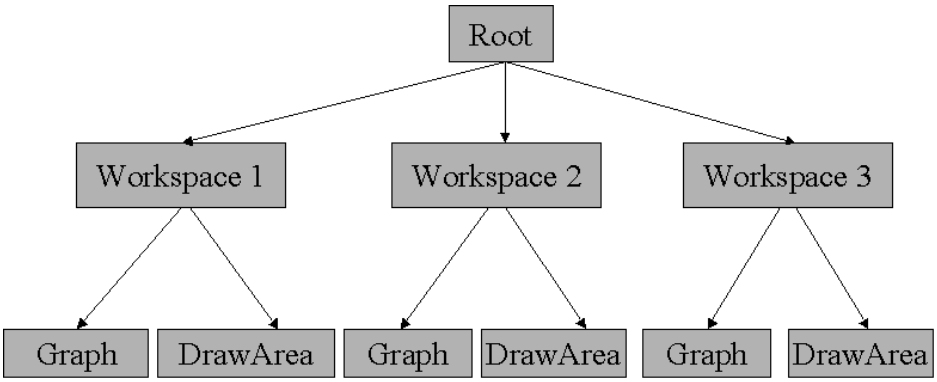

Figure 1 - Example of a MatchMaker synchronization tree

Every client is able to register several listeners for the whole synchronization tree or even for parts of it. A listener class just has to implement the *SyncListener* interface that has the following signature:

```
void objectCreated(SyncEvent event);
void objectChanged(SyncEvent event);
void objectDeleted(SyncEvent event);
void actionExecuted(SyncActionEvent event);
```

These methods are called if there are any changes in the part of the synchronisation tree the listener is registered for. The difference between a *objectChanged* and an *actionExecuted event* is that normally it is not necessary to submit the whole object if there are just minor changes in it. Therefore it is possible to define certain actions that can be performed on an object, by this the possibility of changing objects with a smaller amount of network traffic is achieved.
On the client side every MatchMaker client has several methods to control sessions like creating, joining and leaving sessions. Furthermore the client has the possibility to register and remove listeners to the synchronization tree. Beside this each client is able to affect the whole synchronization tree. Therefore each client has several methods:

```
SyncLabel createObject(SyncLabel parentLabel,Serializable object);
SyncLabel createObject(SyncLabel parentLabel,Serializable
object,boolean sendToAll);
void changeObject(SyncLabel label,Serializable object);
void changeObject(SyncLabel label,Serializable object,boolean
sendToAll);
void deleteObject(SyncLabel label);
void deleteObject(SyncLabel label,boolean sendToAll);
void execAction(SyncLabel label,String action,Serializable
argument);
void execAction(SyncLabel label,String action,Serializable
argument,boolean sendToAll);
```

Those methods communicate with the methods that are provided by the *SyncListener* interface. Every method of the *SyncListener* interface has two corresponding methods in each MatchMaker client. Both of them just differ in one parameter that is a *boolean* flag called *sendToAll*. If this flag is set to *true*, what is also the default behaviour, than the

events according to this method call are also send to the client that performs the method call itself. Figure 2 visualizes the difference between those two possibilities.

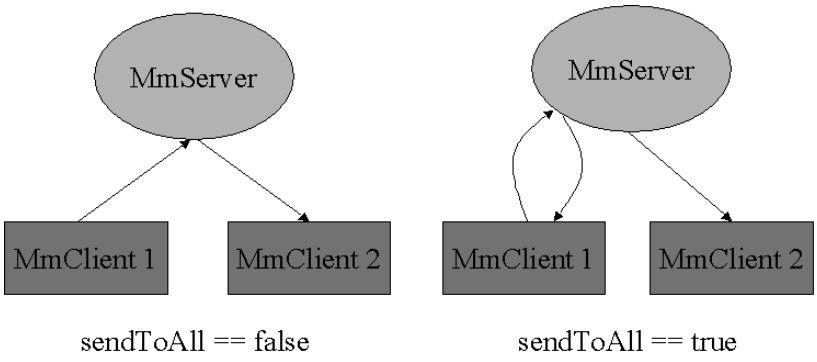

Figure 2 - The difference between sendToAll == true|false

Since MatchMaker is based on the Model-View-Controller architecture what means that it only couples the models of objects. In these so called MatchMaker models all information is allowed as long as it is serializable, for example also information about the view of an object is allowed. This gives the decsion about the visualisation of the object to the developer. For example in an application running on a handheld computer it might be important that the objects are displayed smaller than on a normal computer.
In other applications it might be necessary that not every user gets the whole information about an object. The way the object is visualized is completely up to the developer. In contrast with the MVC the MmModel does not only have the business logik of an object but it might also have informations about the view as long as the information is serializable.

2. The Logging Mechanism

An advanced logging mechanism is necessary for a lot of functions in a collaborative environment, for example if a user should be enabled to replay sessions as well as for the possibilty of undoing and redoing actions. This logging mechanism is realized by adding a listener to the ROOT element of the MatchMaker synchronization tree. This additional listener just writes all actions in a logfile, what could easily be implemented with the help of the four methods from the *SyncListener* interface. The decision to write this logfiles in XML is based on portability issues. This guarantees that, with the help of XSL, every third party application supporting external input is able use the data we collected in our sessions. An example of such a logfile may look like this:

```
<SyncAction action="objectCreated" label="//4" time="1032435449194"
objectType="SyncLabel" user="unknown" number="0" ...>
    <BINARY encoding="base64">
      <!--
rO0ABXNyABxjb20uc3Bpcml0ZWFtLnN5bmMuU3luY0xhYmVsLxBhk/39pa0CAAFMAAV
sYWJlbHQA
EkxqYXZhL2xhbmcvU3RyaW5nO3hwdAABLg-->
    </BINARY>
</SyncAction>
```

The DTD (Document Type Definition) that defines the format of the logfiles looks like this:

```
<!ENTITY % java_dtd PUBLIC "" "java.dtd">
```

```
%java_dtd;
<!ELEMENT SyncActions (SyncAction*)>
<!ELEMENT SyncAction ANY>
<!ATTLIST SyncAction action       CDATA #REQUIRED
                     label        CDATA #REQUIRED
                     time         CDATA #REQUIRED
                     objectType   CDATA #REQUIRED
                     user         CDATA #REQUIRED
                     number       CDATA #REQUIRED
                     typeOfAction CDATA #REQUIRED>
```

This DTD defines that the logfiles have *SyncAction* elements that are children of a *SyncActions* element. Each SyncAction element has certain attributes like a timestamp, a user name, a label, an object type and so on. As a child the *SyncAction* may have any element. Not defining the children of an element is necessary here because we do not actually know all possible objects that could be transferred with the MatchMaker framework, the only thing we know about them is that they implement the *Serializable* interface.
When logging a specific action the logging client tries to write the object itself as an XML child of the SyncAction element it belongs to. Since it is not necessary that every object is capable of writing itself into XML an alternative to XML must be possible therefore the logging client writes the object as a binary element if it notices that the object itself is not able to be written in XML.
Like in Zumbach [8] the logfiles later on can be used for example to analyze the collaboration by the recognition of groups working together, turn taking analysis or punctuation of the processes.

3. The Replay Function

Based on the logging mechanism the MatchMaker framework has a replay function that enables the user to have a look at sessions after they took place. The framework for this replay function consists of a player as a graphical user interface shown in Figure 3 and an extended MatchMaker client that is capable of reading the logfiles and feading the collaborative actions back into the server. Replaying the actions the user has got two possible modes: On the one hand the user can replay the actions time oriented (in real-time or fast-foward) and on the other hand the user can replay the actions step wise if, at runtime of the original session, steps were defined. Those steps are defined by the number of each SyncAction element as defined in the DTD above. All SyncActions with the same number belong semantically to one step. This step wise replay might be interesting for presentations and the time oriented replay, even with fast-forward, is more interesting in learning scenarios. This enables students to replay the whole session, that might be a session of a course, just in the time they need to understand it. Furthermore it is possible to stop or pause the replay if this is useful.

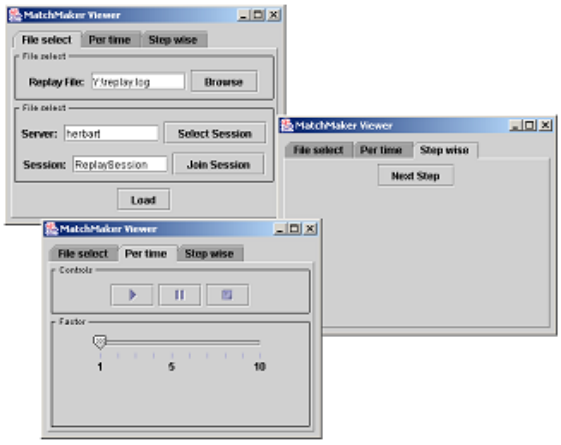

Figure 3 - The user interface for the replay client

4. Undo/Redo Functionality

An additional function of the logging mechanism is to build up stacks that enable the user to undo/redo collaborative actions session wide. If an action occurs the changed part of the synchronization tree is stored. Table 1 shows the processes that are done for any of the four possible events. Only in the case of an *actionExecuted* event first the check whether this action should be undoable or not is performed. This check is necessary because the *actionExecuted* event is normally used to cause very small changes in a collaborative object but mostly not every little change is meant to be undoable. For example painting a stroke leads to changing the stroke element each time a new point is added to the stroke but an undo event on a stroke would be suspected to remove the stroke, not only the last point of it.

Action	***Processes***
objectDeleted	The deleted part of the synchronization tree is stored.
objectChanged	The changed part of the synchronization tree is stored.
objectCreated	The old part of the synchronization tree is stored for which a new child is added.
actionExecuted	Check whether this kind of action should be undoable, if so store the changed part of the synchronization tree, if not, forget this action.

Table 1 - The SyncListeners methods for the logging client

Another example would be the playback of music file: Undoing while playing a music file should lead to stop playing this file, so the start action should be undoable but not the changes needed to play the sounds. Therefore in the MatchMaker framework it is possible to select for every *SyncActionEvent* whether it is undoable or not while by default is not undoable because this fits most situations best.

5. Future Work and Conclusion

Most shared workspace applications that support collaboration use unflexible strategies to achieve the collaboration support. The first and very easy way to couple applications is to use Netmeeting but thats at the same time the most unflexible way because Netmeeting only allows to couple user interfaces what leads to a WYSIWIS (what you see is what i see)

result. The second way for supporting collaboration in shared workspaces is to build own collaboration software. This is for example done in the Belvedere system[9]. Another way is to use already existing frameworks like Habanero[1][10], JavaSpaces[2] or JSDT[3]. The advantage of the usage of MatchMaker is that with this framework it is also possible to couple applications on a semantic level since this framework is based on the Model-View-Controller architecture. Furthermore our framework is very flexible by the grade of the coupling for example it is possible in a shared workspace environment to have private and coupled workspaces as well as only a few layers on the workspaces coupled.
Future development of the MatchMaker framework will focus on transactions, security and server-to-server communication. Transactions will be necessary to ensure the validity of the servers synchronization tree especially with conflicting update actions. For security issues a framework for asynchronous cryptography will be developed that supports both, RSA [5] and ECC [6] with respect to the fact that other encryptions should be easily plugable. The server-to-server communication will be done with SOAP [7] as the underlaying protocol with the benefit of easy communication also through firewalls.

Acknowledgements
This research has been supported by grants of the German Science Foundation (DFG) to Ulrich Hoppe (HO2312/1-1) and by the European community with the COLDEX project (IST-2001-32327).

References

[1] Jackson (1997): "NCSA Habanero Java Object-Sharing Collaboration Framework", In: JavAus97 Proceedings
[2] Freeman, Hupfer, Arnold (1999): "JavaSpaces Principles, Patterns and Practice", Addison-Wesley Pub Co, ISBN: 0201309556
[3] JSDT: <http://java.sun.com/products/java-media/jsdt/index.html>
[4] Paolo Ciancarini, David Gelernter (1992): "A Distributed Programming Environment based on Logic Tuple Spaces", In: ICOT (Fifth Generation Computing) 1992 (Final conference)
[5] Rivest, Shamir, Adleman (1978): "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", In: Comm. of the ACM, 21(2), 1978
[6] Koblitz, Menezes (2000): "The state of eliptic curve cryptography", In: Designs, Codes and Cryptography, 19 (2000), 173-193
[7] SOAP: <http://www.w3c.org/TR/SOAP>
[8] Zumbach, Muehlenbrock, Jansen, Reimann, Hoppe: "Multi-Dimensional Tracking in Virtual Learning Teams an Exploratory Study", In: Proceedings of the International Conference on Computer Supported Collaborative Learning CSCL2002, Boulder, CO
[9] Paolucci, Suther, Weiner: "Belvedere: Stimulating Students' Critical Discussion." In: CHI95 Conference Companion, May 7-11 1995, Denver CO, pp. 123-124.
[10] Soller: "Supporting Social Interaction in an Intelligent Collaborative Learning System", In: International Journal of Artificial Intelligence in Education, (2001), 12, 40-62