"MatchMaker"

Synchronising Objects in Replicated Software-Architectures

Frank Tewissen¹, Nelson Baloian², Ulrich Hoppe¹, Erich Reinberg²

¹University of Duisburg, ²University of Chile
tewissen@informatik.uni-duisburg.de

Abstract

In this paper, we present an approach and an existing implementation to support the object-wise synchronisation of general software entities. The software toolkit “MatchMaker” allows for synchronising already existing software applications by plugging in transparent event listener mechanisms that distribute information via a central server to other remote applications. Based on the original approach that has been presented in 1995, the current implementation of “MatchMaker” serves as the communication basis in real world applications of three computer-integrated classrooms in Europe. Current developments are oriented towards the integration of promising technologies in the field of distributed computing. The main goal of “MatchMaker” is the provision of powerful yet easy applicable high-level API for synchronising distributed objects.

1. Introduction

The need for a software library for synchronising user interface objects originated as an urgent necessity from developing systems for supporting synchronous and collaborative face-to-face learning activities in an electronic classroom. In the original electronic classroom the teacher has an electronic board (such as a Xerox LiveBoard or a Smartech SmartBoard) and every student has a computer. The teaching and learning methodology we had in mind was a succession of presentation, individual problem solving, collaborative problem solving, discussion and group brainstorming [4]. In this scenario, the flow of information between humans but also between the software applications is more communication or process oriented than product oriented [17]. We found out that coordination as well as communication requirements between students and teachers and among students themselves can be easily achieved by synchronising user interfaces of the applications used by students and teachers to accomplish their collaborative work.

Traditional application sharing systems are different from the architecture that is presented here, in that these systems are not replicated. In application sharing systems, usually a server process (master) owns the data that may be shared (i.e. visualised) by clients (slaves). Clients therefore only get a view on the common data, they never physically own the information. This has serious drawbacks for the stability and flexibility of such systems: if the master crashes for some reasons, the information is lost. Furthermore the structure of data that has to be distributed over the network has a loose relation to the real semantic data.

However, the student-teacher coordination often requires only some elements of both interfaces to be synchronised and only for a limited period of time. For example, the teacher may use an application for showing the principles of function derivations with full functionality for entering and displaying any function as well as automatic computation of the function’s derivative (see Fig. 1). The student’s application may allow only to display functions from a given list. Therefore the general idea of synchronising user interfaces a) partially and b) only for a part of an application’s lifetime came up.

While the original approach has been taken up in new versions of the software library, some applications were already in everyday use (e.g. a collaborative whiteboard, see [6]) while others were specially adapted for psychological studies of shared problem solving tasks [12]. The latest Java “MatchMaker” (JMM) version is a complete reimplementation that uses Java specific features (RMI, Reflection, Dynamic Class Loading) and common object oriented design principles [14]. JMM is a development basis in the European Esprit project NIMIS (Networked Interactive Media In Schools, cf. [5]) in the scope of ESE (Experimental School Environments). Current enhancements are mainly oriented towards integrating and taking up new standards in the field of distributed computing, thus being more compatible and more standardised.
2. Principles

The different technological bases for synchronous distributed software environments to be used in a cooperative or collaborative way can be split into three major groups:

- specific application-dependent synchronisation (e.g. IRC chat-tools, network games)
- partial entity-related synchronisation (e.g. JMM)
- general application-sharing synchronisation (e.g. Microsoft NetMeeting, Sun ShowMe)

For the first group of applications, the technical communication is highly application and thus semantic dependent, i.e. it is usually very efficient and flexible within the framework of an application but also it is very specialised with a very limited scope of potential reuse (high programming effort).

The second group tries both, to be general enough to minimise the necessary effort of software adaptations (reuse) and to be specific enough to provide a means for all necessary tasks. The flexibility increases in the case of the tools that synchronise distributed and replicated software environments, but also the complexity of these systems increases (low / medium programming effort, toolkit complexity issues).

For the last group of applications, the technical communication is almost completely application independent, i.e. synchronisation takes place on a very low level, semantic information content is low, requirements for network bandwidth are high but it requires no additional programming effort.

The choice for a system supporting synchronisation in one of the meanings above has to be made in the boundaries of low / high performance and low / high programming effort and the architecture should reflect the actual needs of users and programmers. Additionally the aspects of robustness, flexibility and independence come into play when using these environments in non-laboratory situations (long term servers, unreliable client structure potentially spread over the internet).

Our general approach is a compromise within the given boundaries, whereas the aspect of reuse (i.e. low programming efforts) is one of the most important goals. We accept the limitation of programming language-dependence and of being not able to synchronise a huge set of existing applications (like windows-sharing systems do) but have designed a library that is generic, robust and efficient.

2.1. Architectures

The topic of distributed computing has gained a tremendous amount of interest in the last years, particularly due to the growth of the internet. Still, there is no standard mechanism to synchronise or “plug-in” synchronisation facilities to existing distributed architectures. Applications associated with trendy buzzwords like e-commerce or WAP but also many interesting specialised and portable devices like PDAs are still struggling with one of the main problems of distributed computing, standardisation. There are related process communication approaches that focus on the language that is used to communicate with other processes (proprietary solutions like DDE for PC/Windows systems or KQML as a general “agent” communication language). We are interested in architectures that provide a complete framework for distributed computing, taking aspects of process lifetime, naming services, interface conventions, etc. into account. Furthermore the aspect of non-intrusive extensions of existing software applications and the approach of dynamic coupling and de-coupling of software entities plays an important role.

Some architectures, approaches and software libraries already provide solutions that are different on the level of complexity, generality and flexibility:

**CORBA** provides an infrastructure that enables invocations of operations on objects located anywhere in a network as if they were local to the process using them [16]. The basic mechanisms to provide this functionality are uniform definitions of interfaces by means of IDL (Interface Definition Language) and ORBs (Object Request Brokers), i.e. by defining interfaces in a common language-independent way and by providing a network of interoperable servers. The functionality of ORBs and the communication between each other are standardised and the way different aspects of server objects (lifecycle, naming, persistency) are handled is in a uniform way. Although many elements of CORBA are signposts in the field of distributed computing, it is a complex software architecture and the level of API that is presented to an application programmer very much depends on the programming language and additional support libraries.

**HABANERO** is an implementation that supports collaborative environments by transferring objects from one environment to another [8]. The basic means for synchronisation is the transfer of object states (serialization) and the distribution of Java events. Compatibility is provided by a session manager (Habanero server) that is responsible for ensuring that classes are compatible and that the initial state of an object is transferred to an object that is about to join a session, etc.. Subsequent changes of an object’s state are distributed based on Java’s event queue mechanism and associated listener methods. Additionally Habanero provides a user interface for accessing information about participants, connected applications and alternative servers. Habanero very much depends on Java’s basic event stack, same class instances on the distributed client hosts and it has to deal with some basic problems of distributing predefined events (e.g. the event-source side-issue). To make an existing application or applet collaborative, additional
methods have to be implemented (a conversion wizard is provided) and the class hierarchy changes during the so-called “habanerisation”.

**VOYAGER:** Parts of the commercial product family “Voyager” represent a promising approach that combines different edges of the most prominent technologies [11]. Voyager takes up general concept of CORBA and adds even more flexibility (i.e. CORBA as such is also a special case of communication) by enabling access to other special services (e.g. RMI, DCOM). On one hand, the systems tries to be as flexible as possible in terms of compatibility with existing implementations (under the notion of “non-intrusive” toolkits) and communication protocols, on the other hand the core API is still easy to use. Synchronisation has to be implemented explicitly, but with low programming effort and embedded in a rich set of features (like e.g. as a platform for mobile “agents”).

**JAVA SPACES** technology takes up the approach of “tuple spaces” and offers spaces as shared networked-accessible repository for objects [3]. Basic read, write and locking mechanisms ensure, that the information stored in such a space is consistent and mutually available for all clients. Clients that want to communicate with each other do not need to have specific information about each other, they just share a common space and access the space with templates of structured information (including wildcards for specific sub-information pieces). Although this is a simple but powerful mechanism for Java, client processes implicitly need information about the meaning of a information and furthermore about the type (i.e. a specific Java class in this case). For example a client might lookup the velocity information of an aeroplane expecting a number (e.g. a long value) and may never get a result, because the information was stored as a three-dimensional velocity vector.

**MATCHMAKER:** In contrast to the approaches and systems mentioned above, JMM mainly differs in two aspects: The approach of JMM is to synchronise existing software entities in existing, distributed and different applications. As mentioned earlier, the view on distributed systems in face-to-face situations is more communication-oriented than product-oriented. This object-wise late-binding strategy has major impacts on the way a potentially collaborative application has to be built and on the runtime-flexibility available for the user.

JMM is actually a fairly simple system, i.e. programmers can start from very simple examples and small coupled entities and proceed to complex application synchronisation by plugging together a set of simple entities. This goes very much along with the general approach of object oriented programming and keeps the basic mechanisms simple and maintainable. Admittedly, JMM does not help with the general problem of standardisation, but it throws light on the most promising directions from its own background, thus adding a piece to the puzzle of requirements for integrated technologies in the future.

Because from our point of view (focused on face-to-face situations and the need for highly flexible and “light-weight” toolkits) the “race” of concurrent approaches and technologies appears still open, we present our approach and its implementation that has successfully been applied for some years now.

### 3. Framework and Existing Applications

Based on the requirements for face-to-face situations, synchronisation methods, software toolkits and libraries should have the following characteristics:

- **Replicated architecture:** It should be possible to synchronise multiple and potentially different applications already existing in a distributed environment.
- **Dynamic synchronisation:** It should be possible to start and stop the synchronisation at any point of the application’s lifetime. Before and after the coupling phase, applications should continue to exist independently.
- **Partial synchronisation:** It should be possible to synchronise each component of an application’s interface individually with a component of another application’s interface, thus allowing the synchronisation of applications with completely different interfaces.

Because these synchronisation requirements are common to many applications, it seems interesting to develop a generic solution. The generic solution was the development of a series of programming libraries allowing programmers to simply develop distributed applications based on the “coupled objects” philosophy.

#### 3.1. Development phases

A first attempt to develop a library allowing the programming of interfaces with coupling and de-coupling facilities was done in 1993 at the GMD-IPS institute in Darmstadt, within the research activities of the COSOFT (COMputer SuppOrt for Face-to-face Teaching) research group (cf. [2]). The aim of the research was the development of teaching/learning models and systems for supporting teaching/learning activities in the electronic classroom. The electronic blackboard used by the project was a prototype of the Xerox’s LiveBoard, which was based on a SUN-Unix workstation. In this version, coupling and communication facilities were added to an existing library developed with the aim of making the development of applications with graphic interfaces easier. This library was written in C and introduced new widgets which were easier to handle by wrapping the original Motif widgets. This library was extended by including some new functions to couple and de-couple the new widgets. Communication between applications was
based on a client-server model. Applications which were going to be synchronised should register first with the server with a distinctive name. Coupling and de-coupling of objects was pair-wise and could be achieved by executing calls like `CoCouple{obj1, obj2}` and `CoDecouple{obj1, obj2}` respectively. Naming of the objects was hierarchical, being the application’s name the first component in the hierarchy. Coupling was also hierarchical, which means that by coupling a “parent” object (container), all the contained object were also coupled. The library also provided a RPC facility: a callback function of a widget of any registered application could be invoked from another registered application.

The last predecessor of JMM was a C/C++ version of MatchMaker, developed at the COLLIDE research group in Duisburg. Still centred around the topic of coupled user interface objects, this new version has been successfully used in e.g. [10] and in [15] as one basis for software implementations in a Computer Integrated Classroom scenario.

3.2. Existing Applications

From the set of existing applications that use the JMM functionality as a means for collaboration, we present two prototypical applications. Both are similar in that they can be partially and dynamically synchronised.

Fig. 1 shows two instances of one of the early collaborative applications (based on the former C/C++ version of MatchMaker). In this scenario two different versions (a teacher version in the background of Fig. 1 and a student version) are used to present, discuss and practice the topic of functions and derivatives in the field of mathematics. In the scenario of a computer integrated classroom where the teacher needs to discuss a student’s solution for the public, it turned out that a full application or window synchronisation is not desirable. Instead, the teacher initialises the synchronisation by coupling the function input fields (i.e. the generator of the displayed curve, not the display itself). At any point in time the applications can be de-coupled, e.g. to synchronise with another student, or to synchronise with a larger group of students to distribute a special task.

Fig. 2 shows some users with a recent application of JMM. Two children work in an collaborative mode with two instances of a Java-based application for early literacy. The material (a table built up of images and letters) is distributed between the participants based on the jigsaw design principle to induce collaboration. The actual workspace is coupled by means of JMM. Two or more workspaces of this application can be coupled and de-coupled by the children according to their needs and preferences. The application is in daily use in a German primary school associated to the NIMIS project [5].

4. Challenges and Requirements
After a process of some design and re-design cycles based on different operating systems and programming languages, one of the most interesting challenges is to fully support all of JMM’s features for already existing Java applications and user interfaces.

4.1. Technical Challenges

The development of JMM poses some interesting design and implementations problems to solve. The first one is the problem of object locking in distributed environments [18]. JMM’s approach does not imply a central repository of data that can be accessed from different clients, but relies on sets of many objects that exist in different applications, synchronised only by the distribution and replication of events generated by an object-member of a set. Depending on the “network distance” between applications, this procedure may be completed almost instantaneously or it may have a considerable delay, which is more likely to happen when some long term actions (e.g. drag and drop actions) are being carried out. In such cases another event may be triggered before the distribution of the first one is completed. The locking (inhibition of event triggering) of all the coupled objects during the procedure is not a plausible solution because the locking message takes also time to reach all the objects. Implementing an event queue for each group of coupled object may solve some problems but it will demand a greater programming effort and the event distribution speed will be much slower. Taking this into account, we experienced that a good strategy is not to care about the locking problem at all, not at least because there are many situations in which any order of events causes the same effect.

Another interesting problem is the coordination of the state of the coupled objects. For example, if we have two or more text areas which we would like to couple, only those characters typed after the coupling will be distributed. If the content on both text areas was different before the coupling, the objects will remain in a different state (that means with different content) despite the coupling. A first solution to this problem is forcing the state of one of the objects into the others, so all participants start with the same state (in the current JMM implementation, the initial states of objects are synchronised according to this rule). On the other hand, sometimes one would like to have the opposite state (or more complex relations) in two coupled objects, like in the case of a radio button which may give the exclusive right to talk or write in a video conferencing situation. Therefore, a more general solution should have an explicitly accessible function which will transform (i.e. not only copy) the state of one object, providing the programmer with more control over the initial state of a set of coupled objects.

The last interesting challenge that we want to mention here (which is not only a technical one) is the naming problem. Every time a software entity wants to communicate with another entity it needs a reference. Inside an application or even on the same host it is easy to use something like memory-references (pointers) or unique identifiers. When leaving the cage of one machine and potentially entering the internet world, the question is “How to address a remote entity?” (syntactic level) and “How to find out what it does?” (semantic level). On one hand, JMM has a clear strategy to solve the first problem via URL-style addresses (see section 4.3.) which have developed as the standard way to address potentially distributed entities. On the other hand, JMM of course does not solve the second problem. Here, only a standardisation of functional meanings on a meta-level can help. Currently the JavaSpaces / Jini direction seems to be the most promising approach to deal with these kind of additional meta-information.

4.2. Requirements: The Programmer’s View

Programmers’ experiences and implementation aspects of the JMM toolkit shed a specific light on requirements of software libraries for synchronised objects. The following list gives an overview on how the current state of JMM relates to these aspects:

**Flexibility:** A flexible library for synchronising objects from the programmer's point of view should be non-intrusive, i.e. the coupling of existing applications and objects should be simple. In that sense, JMM provides a high degree of flexibility because it provides standard behaviour for many classes (i.e. no additional programming effort) and a rich set of powerful basic classes that can be easily extended to the special needs of an application. The coupling itself is always straightforward and simple.

**Compatibility:** For a toolkit like JMM, this term refers to a definition of “compatible” objects. Objects of the same type are by definition compatible and couple-able. Furthermore, there are also objects which have not identical signatures (i.e. different classes in the case of Java) but have such similarity in that it makes sense to define a logical coupling protocol between them. A wide range of compatibility were already implemented in the very first versions of the Motif-Toolkit (see [17]). For the Java based versions, flexible compatibility is even more natural: some objects just share the same kind of property / event, i.e. they are compatible.

**Extensibility:** It is obvious that no toolkit can serve all needs, but furthermore it might even not be a good approach to go for a “serves-all” solution. The biggest flexibility if offered by well designed APIs embedded in an object hierarchy that makes quick starts easy. This is one major success criteria of the Java language and Sun’s development kit. JMM provides flexible and easily
reusable APIs for application programmers that just use JMM’s existing functionality, but also for programmers that are going to extend the library and add custom behaviours. Both ways of usage have been applied by different programmers with varying background during the last years and the library was continuously adjusted to their needs. While JMM is being used in the EU-Project NIMIS (see [5]), no basic change was necessary.

**Standardisation:** Although there is a clear need for standardising the syntactic and semantic aspects of distributed software systems, JMM, as many other libraries is not standardised. Nevertheless, JMM refers to a standardised philosophy and uses standardised tools. Two threads of activities will try to improve the general standardisation of JMM (see section 5.).

### 4.3. Existing Functionality and Architecture

The current version of JMM supports a wide range of standard user interface objects (i.e. all AWT-Components, most of the Swing-Components) and high-level APIs for custom class coupling, application properties and flexible remote procedure calls. The exact definition of the coupling functionality is the following:

**Given an instance a of class A and an instance b of class B, a coupling call succeeds if A and B are compatible (compatibility), A and B exist in the local Virtual Machine (VM) or exist in remote VM and are bound to a symbolic name (accessibility) and A and B exist in a VM that is accessible (activity).**

**Compatibility:** In the current version of MatchMaker two classes are most compatible if they are equal or compatible in the sense of the highest common denominator if they share same parent classes in their class hierarchy. For example a Button class is compatible to the Label class in the sense that they share common attributes (Dimension, Text, Color) that can be synchronised i.e. coupled. This functionality is associated to the most specific shared parent class (the Component class in this case). Furthermore two classes are compatible if specific designated attributes are declared to be coupled and both objects share these attributes. Therefore non-related classes are compatible if they share attributes (e.g. a property “textContent”, see Fig. 4) that are of interest for the coupling.

**Accessibility:** Objects are accessible for a coupling call in VM A if they reside in this VM A or they reside in a different VM and have a symbolic name. Symbolic names are used to identify and access remote objects from within other processes. Names in MatchMaker follow an URL notation form:

```
//hostname/applicationname/parentname/objectname
```

To be accessible from outside, an object must be bound to a name. Every MatchMaker instance controls a NameService that takes care for potential naming conflicts and effective storage and retrieval of name / object relationships.

**Activity:** An object A is called active or alive if the VM still reacts to incoming network events. Usually this goes along with the responsiveness to user inputs. If an application is not alive in this sense, the application is removed from JMM’s database and all coupled objects within that originated from this application are removed from their coupling sets automatically. After a JMMS instance is launched on a dedicated server host, the basic steps to establish a set of coupled objects are:

- **Create a MatchMaker client instance**
  i.e., connect to MatchMaker server, assigning a unique name to the application and generating a new name space for this application on the server side

- **Bind objects to symbolic names**
  i.e., associate symbolic names with local object references within the namespace of the application

- **Couple two objects**
  i.e., invoke a couple(reference_A, reference_B) call, where a reference can either be a local object reference or a symbolic name to address objects in remote applications

Fig. 3 locates JMM’s functionality in the OSI multi-layer model. JMM itself is located on the so-called Application Layer, whereas the applications built on top of JMM reach a higher degree of abstraction. With the aid of JMM, coupled Java applications can easily implement a functionality that is quite close to the real meaning of a user’s synchronisation request. The example share_workspace-call was taken from a current software application that is shown in Fig. 2.

The actual coupling functionality is encapsulated in special objects (so-called Coupler). To provide high-level APIs for programmers and predefined classes for common standard classes (like the Java Swing and AWT classes), JMM provides skeleton implementations in the form of multi-purpose base classes. Fig. 4 shows the actual message flow and the distributed responsibilities to establish the coupling functionality for the specific case of a TextComponent.

<table>
<thead>
<tr>
<th>OSI Layer View</th>
<th>Typical functions / semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coupled Java Application</td>
<td>share_workspace(partner)</td>
</tr>
<tr>
<td>MatchMaker</td>
<td>couple_objects(object1, object2)</td>
</tr>
<tr>
<td>Presentation Layer (Java RMI)</td>
<td>call_remote_method(arguments)</td>
</tr>
<tr>
<td>Session Layer (Sockets)</td>
<td>open_socket(host, port)</td>
</tr>
</tbody>
</table>

**Fig. 3 JMM’s functionality in the OSI layer model**
The sample code below shows how to setup a basic JavaBean that encapsulates the information of a simple text:

```java
public class CString {
    private String content;

    public void setContent(String newContent) {
        oldContent = content;
        content = newContent;
        pcs.firePropertyChange("content", oldContent, newContent);
    }
}
```

**Fig. 5 A simple JavaBean class**

Because the class uses the general bound property and event mechanism of a JavaBean, no further data or method is needed concerning the coupling mechanism. The coupling therefore may be initialised by a simple couple-call:

```java
mm.couple(cli, c2, "propertychange.content");
// with mm: MatchMaker instance,
// cli, c2: CString instances
```

**Fig. 6 Instantiation of two coupled objects**

In addition to the coupling facilities, JMM features also a flexible access to remote methods (similar to the approach presented in [11] and very recently in [9]). Although Java’s core RMI already provides the basis for this facility, JMM offers an easier alternative for the programmer using the reflection API. JMM provides an easy way to call a (static or non-static) method in a remote class or instance just by giving the symbolic name of a class or a class-instance and the symbolic name of a method. The result is a very flexible architecture that allows for remote processes to call any public method of a class or of an object in another process only with the necessary core information (without the need for additional stub-classes or compilers).

5. Conclusions

In this paper, we presented the basic approach, the historical line of development and the current state of a generic toolkit for synchronising distributed objects in a replicated architecture. Currently, we try to improve features related to the aspect of standardisation and flexibility in JMM in three threads. The first activity thread is centred around new standards, like the JavaSpaces / Jini technology but also with respect to real system solutions, like Voyager. Further developments of JMM will try to integrate or to be based upon this technology without losing the strengths in flexibility and extensibility as a distributed and replicated architecture.
The second thread, on a conceptual level, is centred around new approaches, i.e. ways of thinking, talking and implementing coupled applications. Here, an approach similar to the JavaSpaces technology, but with replicated spaces, is one envisaged extension: Objects are not coupled pair-wise, but join a group of coupled objects by registering to a coupling-party identified by a unique name. The coupling-party is created by the first object trying to join a non-existing coupling-party. In some circumstances this may lead to clearer representation of what is actually the group of coupled objects, thus avoiding the effect of “transitivity” of a pair-wise coupling method. It also reduces the amount of necessary public naming-information because only the coupling party name should be agreed on and made public. A first version of this approach for coupling objects has been implemented by Reimberg ([13]).

The third thread is related to the way the library itself is distributed, accessible and possibly taken up by third parties. Although JMM will be one core part of commercial software products, possibilities of making it more freely available for other interest groups are considered (e.g. as a non-commercial freeware in the community of Java developers).

6. Acknowledgements

Parts of this work refer to the European Esprit project No. 29301, NIMIS. We wish to thank all people who used the library and contributed to enhancements of MatchMaker during the last years. Special thanks to Jian Zhao, Katrin Gassner, Martin Mühlenbrock, Rui Prada, Andreas Lingnau, Volker Witzel, Eric Felhse and Rolf Plötzner.

7. References