

# Massively Distributed Virtual Worlds a Framework Approach

## MaDViWorld: a Java Software Framework for Massively Distributed Virtual Worlds

Patrik Fuhrer and Jacques Pasquier-Rocha

University of Fribourg  
Department of Informatics  
Rue P.-A. de Faucigny  
CH-1700 Fribourg  
Switzerland

`patrik.fuhrer@unifr.ch`

WWW home page: <http://diuf.unifr.ch/~fuhrer/>

**Abstract.** The aim of this paper is to briefly present the general concept of virtual worlds and then to focus on distributed and decentralized ones. MaDViWorld is a Java framework for massively distributed virtual worlds. We first present its software architecture and then discuss some of its specialized features, namely: the object structure, the distributed event model and the lookup mechanism for the rooms. These are the main aspects that evolved since the first version of the framework. To conclude, some example objects and further potentialities of the framework are discussed.

Keywords: Software Framework, Virtual World, Distributed Events

A shorter version of this paper was presented at the **FIDJI'2002 International Workshop** on scientific engineering of Distributed Java applications, November 28-29, 2002, Luxembourg University of Applied Sciences (IST), Luxembourg-Kirchberg, LUXEMBOURG and has been accepted for publication in the *Lecture Notes in Computer Science* by ©Springer Verlag.

## 1 Introduction

### 1.1 The Virtual Worlds Paradigm

The document paradigm is well-known in today's Internet technology: documents are made available on one or several servers and client applications (e.g. Web browsers) are used in order to interact with them. The underlying metaphor is the one of a huge cross-referenced book where each user browses through the pages totally unaware of other users performing the same task at the same moment. All actions are asynchronous and, thus, there is no need for a central server to coordinate user interactions with the pages of the book or to take care of an event redistribution mechanism.

Within the *virtual world paradigm*, multiple users and active objects interact in the *same space*. Therefore they have a direct impact on each other. Within such systems, if a user interacts with an object, the other connected users can see her and start a dialog with her. Moreover, it is possible for a user to modify some properties of the world and all the other users present in the same subspace (e.g. the same room) must immediately be made aware of it. Examples of the virtual world paradigm range from simple graphical chat to sophisticated 3D virtual worlds used for military simulations.

For a good comprehension of the present paper, the following four terms need to be briefly explained:

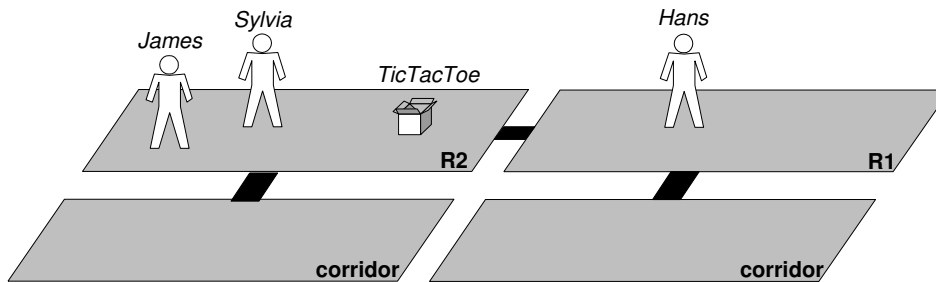
1. *Avatars* are the virtual representation of the users. Concretely, an avatar is a tool that allows a given user to move through the world, to interact with its inhabitants and objects and that lets the other users know where she is and what she is doing.
2. In order to distinguish between near and distant elements it is essential to divide the world into subspaces where the users might or might not enter and in which all interactions take place. Otherwise, the world would not scale. We call such subspaces *rooms*.
3. Rooms are connected by *doors*, which an avatar can use for moving from one room to another.
4. *Objects* populate the rooms and they can be either passive, reactive or active (see [15]). Furthermore, in a distributed world, it should be possible to "physically" transport a given object from one room to another.

For a more detailed terminology and a historical overview of virtual worlds, the interested reader can refer to [6].

The conceptual model, that emerges from these considerations, is shown in Figure 1. It represents a very simple world with four rooms, three avatars (James, Sylvia and Hans) and a single game object (TicTacToe). One can also see how the rooms are interconnected by three doors.

### 1.2 MaDViWorld Goals

The main advantage of the document paradigm approach is that it allows a really distributed architecture with thousands of http servers interconnected all



**Fig. 1.** The conceptual model of a simple world

over the world. If a crash occurs, only the pages hosted by the failed or the no longer reachable servers become momentarily unavailable. The whole system is extremely robust and, since the connection of new decentralized servers is always possible, there is no limit to its growth.

At the software architecture level, systems based on the virtual world metaphor are clearly more complex. Indeed, the users directly interact with the original objects of the system and the resulting event must be correctly synchronized and forwarded in order to maintain the consistency of the world. This explains why most of them are based on a client-server model, for which a single server or more rarely a small cluster of servers contain all the world pertinent data and assume the world accessibility, consistency and persistence. On the client side, many of them enable interaction with the other users and the various objects of the world. This approach depends completely on the central server robustness and does not scale well.

Some proposals for completely distributed environments can be found in the following projects: URBI ET ORBI [4], DIVE [5] and MASSIVE [11] or in [3] and [19].

The goal of our research group is to define software solutions in order to support the virtual world paradigm presented above, without making concessions to the single server architecture. Actually, *MaDViWorld*, the acronym of the software framework presented in this paper, stands for Massively Distributed Virtual World, since its subspaces are distributed on an arbitrarily large amount of machines. The only requirement is that each machine containing a part of the world runs a small server application and is connected to other machines through the Internet. This obligation is exactly the same as the one required by the World Wide Web document paradigm approach, with all its advantages in terms of robustness and scalability.

## 2 MaDViWorld Software Architecture

First of all, it is important to notice that it is out of the scope of this paper to explain the framework in all its details. This paper explains the concept of using

a framework approach at a rather high level of abstraction and then concentrates on some critical points, that have been solved.

Most diagrams used to illustrate the framework are designed as UML class diagrams, or as UML sequence diagrams. More information about the UML notation can be found in [8].

More detailed information is available in [6] and in [7], which describe the first version of the framework. Furthermore, an Object Programmer's Guide is reachable from the official MaDViWorld web site [13]. The guide includes a tutorial for developing new objects. The working framework can also be downloaded, with an installation guide and the Javadoc of the framework.

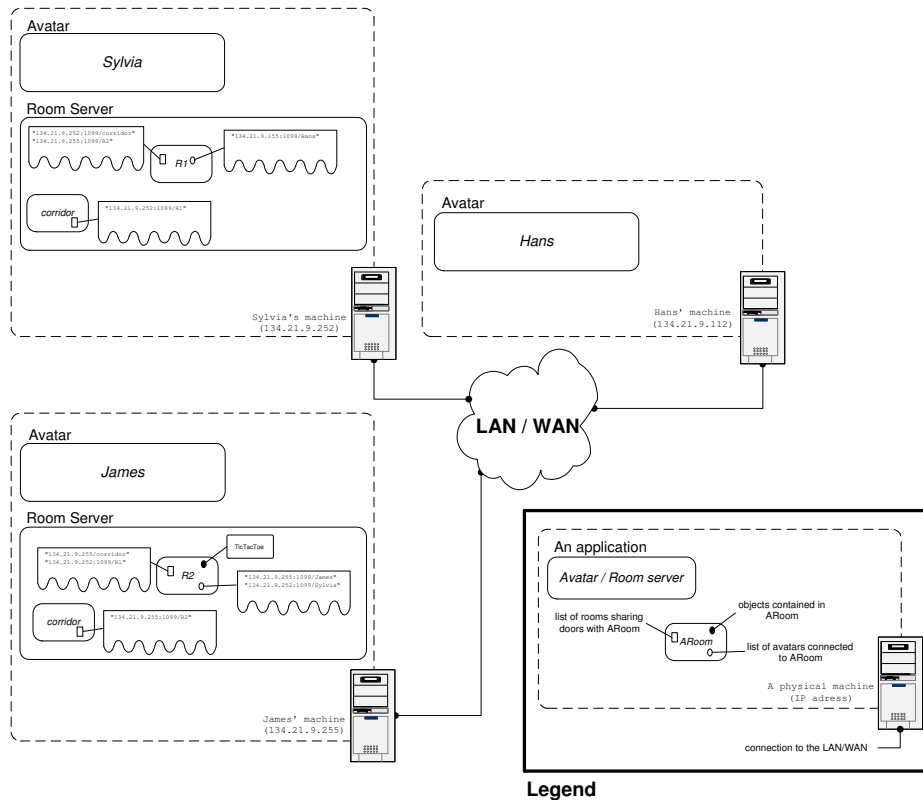


Fig. 2. One possible physical model for the conceptual model of Figure 1

## 2.1 Massive Distribution Explained

In order to clarify what is meant by *massively distributed*, let us reconsider the conceptual model of the simple world presented in Figure 1. As already mentioned, there must be no central server but arbitrarily many of them and none of them has to know the whole world. Figure 2 shows a possible physical model, illustrating what concretely happens at runtime. Room R1, one of the two corridors and Sylvia’s avatar run on Sylvia’s machine, while Room R2, the other corridor and James’s avatar run on James’s machine. Finally Hans’s avatar runs on Hans’s own machine.

## 2.2 The Framework Approach

There are a lot of available definitions for what a framework is. A complete and precise one is given in [16]. But let us take the following more concise one that can be found in [2]:

A *framework* is a partially complete software (sub-) system that is intended to be instantiated. It defines the architecture for a family of (sub-) systems and provides the basic building blocks to create them. It also defines the places where adaptations for specific functionality should be made. In an object-oriented environment a framework consists of abstract and concrete classes. The instantiation of a framework involves composing and subclassing the existing classes. A framework for applications in a specific domain is called an *application framework*.

Keeping these definitions in mind, this section describes the MaDViWorld application framework. The different places where adaptations for specific functionality should be made will be identified and explained. These flexible points of the framework are commonly called *hot spots* and are opposed to the *frozen spots* (see [18]). An overview of the whole framework is shown in Figure 3. The root package for all the other MaDViWorld classes is `ch.unifr.diuf.madviworld`, and in the rest of the paper this will be omitted for evident convenience reasons.

First, let us consider an *horizontal* decomposition of Figure 3:

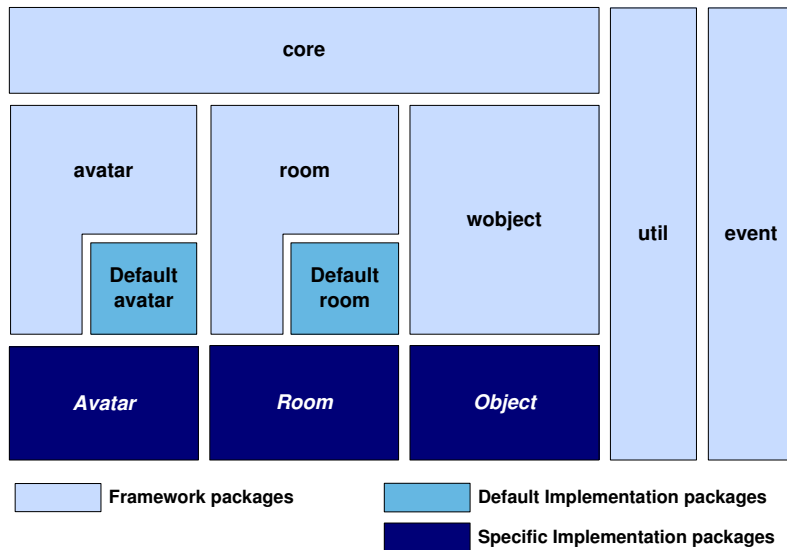
- The communication between components is defined by a protocol. The upper layer of the framework fulfills this task. The framework user only has to understand this visible interface of the components. That’s why MaDViWorld would rather be classified as a *black-box framework* (see [14]). The `core` package encloses all these interfaces.
- The middle layer consists the default implementation packages of the framework. It contains concrete and abstract classes.
- The lower layer, finally, is for the concrete implementation, where all the application specific classes are placed.

Second, let us decompose the blocks of Figure 3 *vertically*. From left to right one finds respectively all the packages and classes relative to the client applications (i.e. avatars), then those relative to the server applications (i.e. rooms)

and those relative to the objects populating the rooms. Ultimately there are two utility packages, one containing packages and classes used by the framework and leftmost the event package. Obviously, the implementation or extension of the *Avatar*, *Room* and *Object* packages are the *hot spots* of the MaDViWorld framework. There are two types of activities for a framework user:

- The first one consists just to enrich the world with new types of active objects by providing the appropriate implementations of the `wobject` package. For this activity, users may use the standard avatar and room server application and use a little wizard application to install their new objects dynamically into a running (potentially remote) room.
- The second one consists to provide richer or better avatar and/or room server applications either by extending the default ones, or by both extending and implementing the appropriate framework extendable classes or by fully implementing the appropriate framework interfaces from scratch (see Figure 4).

The current version of the framework consist of a total of approximatively 100 classes organized in 10 packages and counts 14000 lines of code. If one counts all the already available objects, the total amount of classes is greater than 200 composed of 37000 lines of code.



**Fig. 3.** Overview of the MaDViWorld framework

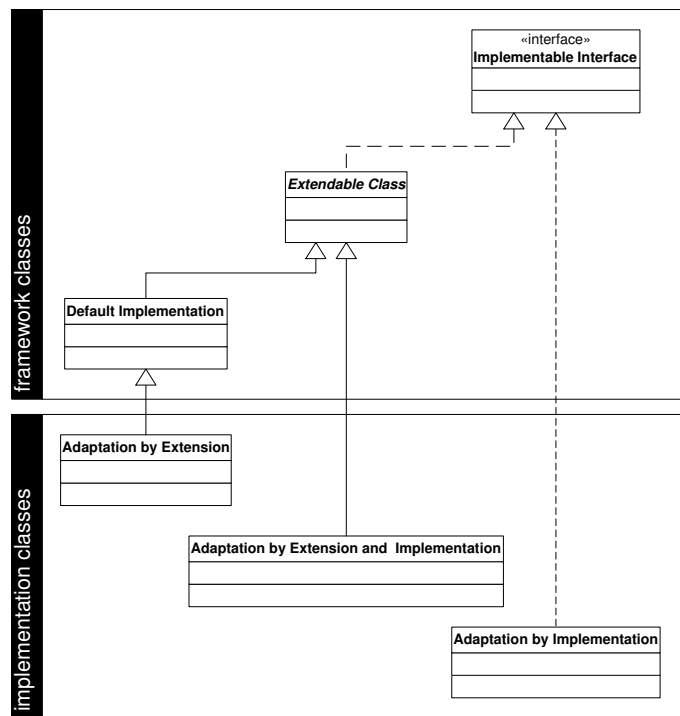


Fig. 4. The three modes of adaptation offered to the framework user

### 3 Objects, Events and Lookup

This section focuses on the objects and particularly on the distributed event model of the framework, which is one of the aspects that was not yet present in the first version of MaDViWorld as presented in [6] and which plays a central role for objects. The third part discusses another point that is new in framework: the integration of JINI technology.

#### 3.1 Object Structure

The purpose of this section is to go over the main points of the objects creation and use. Figure 5 illustrates how the `wobject` package has to be implemented in order to develop new objects. For further comments about this package the reader is invited to consult the Object Programmer's Guide on [13].

In order to add a new object, the framework user has to create the corresponding `newobj` package, which must contain two subpackages, one for the object's *implementation part* and one for its *graphical user interface (GUI) part*. This clean separation between the user interface and the object logic does not provide a two-way communication channel between these two parts. The client relationship between the `MyObjGUIImpl` class and the `MyObj` interface provides a one-way communication channel (from GUI to the implementation), but the implementation part cannot send information back to the GUI. The distributed event model designed to address this issue is presented below.

#### 3.2 The Distributed Event Model

Events play a crucial role in the MaDViWorld framework. Schematically, each time the state of one of the world components changes, a corresponding event is fired by the altering subject and consumed by the registered listeners, which react appropriately. The management of all these events is a complex task:

- They are in reality remote events and several network problems can occur;
- Some of the events have to be fired to only a subset of all the listeners;
- Some listeners may not be interested in every type of event.

Thus the framework must offer a *distributed event model* that handles all these situations.

The two last points listed above, lead to the elaboration of an abstraction for creating unique identifiers. DUID is the acronym for Distributed Unique ID and is implemented in the `core.DUID` class<sup>1</sup>. Each room, object or avatar has an associated DUID that is generated by the framework, so that it can be identified without ambiguity. The use of such a DUID was inspired by [12].

It is now time to take a closer look at the content of the `event` package (see Figure 6 ) and how it solves the mentioned problems:

<sup>1</sup> The DUID is the combination of a `java.rmi.server.UID` (an abstraction for creating identifiers that is unique with respect to the host on which it is generated) and an instance of `java.net.InetAddress` (a representation of the host's IP address where the object was created which makes the UID globally unique).

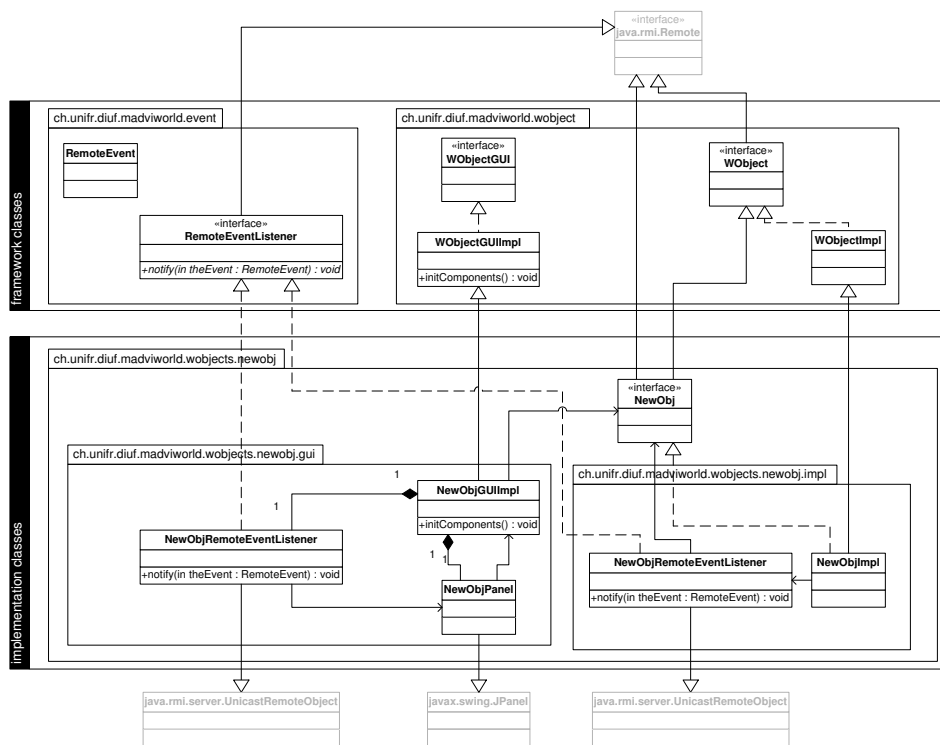


Fig. 5. Implementation of the `wobject` package

- The `RemoteEventListener` interface extends the `java.util.EventListener` interface and defines the single `notify()` method. Any object that wants to receive a notification of a remote event needs to implement it.
- The `RemoteEventProducer` interface defines the methods needed to register, unregister and notify event listeners used to communicate between different parts of the system. The register method takes as parameter the event type the listener is interested in. There are five possibilities: *all events*, *avatar events*, *room events*, *object events* and *"events for me"*. With the latter, the listener is only informed of events addressed explicitly to it (thanks to its DUID), without paying attention by whom.
- The `RemoteEvent` class defines remote events passed from an event generator to the event notifiers, which forward them to the interested remote event listeners. A remote event contains information about the kind of event that occurred, a reference to the object which fired the event and arbitrarily many attributes.
- The `RemoteEventProducerImpl` class implements the `RemoteEventProducer` interface.
- The `RemoteEventNotifier` helper class notifies in its own execution thread a given event listener on behalf of an `RemoteEventProducerImpl`.

Figure 6 shows the *design pattern* used through the whole framework for the collaboration between the three different parts of MaDViWorld (i.e. avatars, rooms and objects) and the utility `event` package. Note that the three of them are both implementing the `RemoteEventProducer` interface and are client of its implementation, `RemoteEventProducerImpl`. The operations defined by the interface are just forwarded to the utility class. With this pattern we have the suited inheritance relation (a `WObject` 'is a' `RemoteEventProducer`) without duplicating the common code. A lot of similarities with the Proxy Pattern defined in [10] can be found.

To sum up the whole event mechanism, the UML sequence diagram of Figure 7 dwells on all the operations, from the registration phase to the firing and notification of an event.

First (a), the event consumer registers a `RemoteEventListener` to a room, avatar or object whose events it is interested in. Second (b), due to a state change an event is fired and all interested listeners are notified, each by a `RemoteEventNotifier`. The informed listener can then do the appropriate work with regard to the type of the event. On Figure 7, one can also see the different methods invoked remotely across the LAN. This pattern present some similarities with the *Jini distributed event programming model*, which is specified in [1] and thoroughly explored in [17].

### 3.3 Lookup

Let us briefly discuss one big difference between the actual and the first version of the prototype framework: it is Jini enabled.

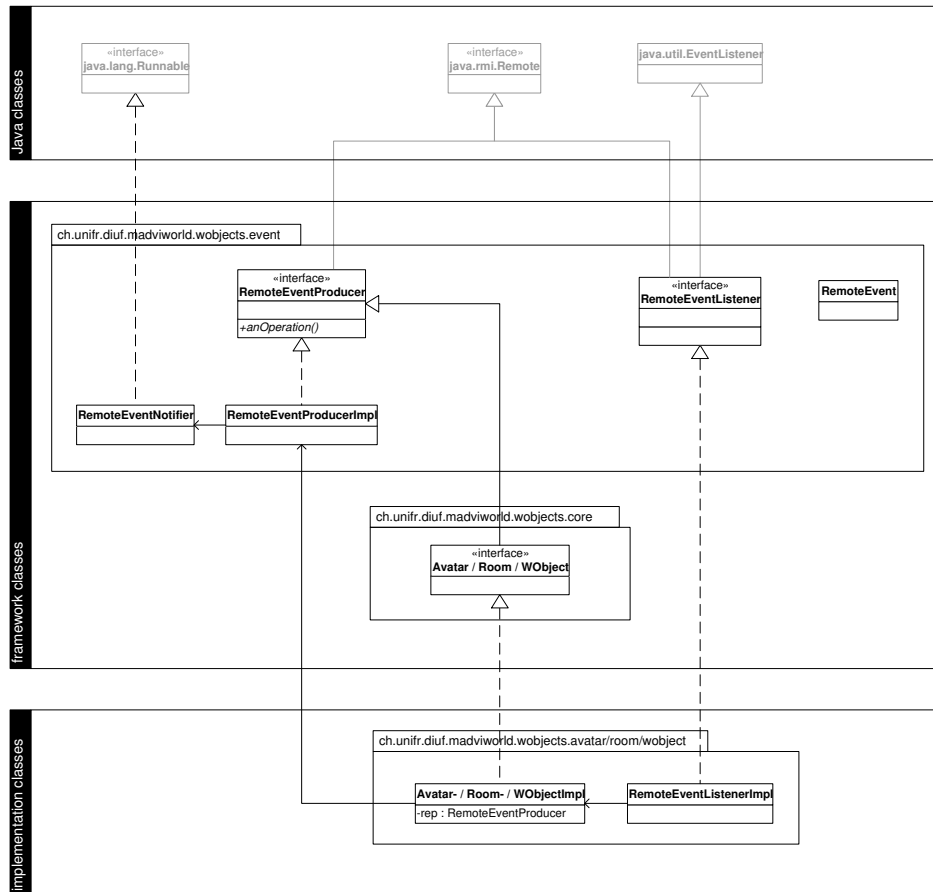
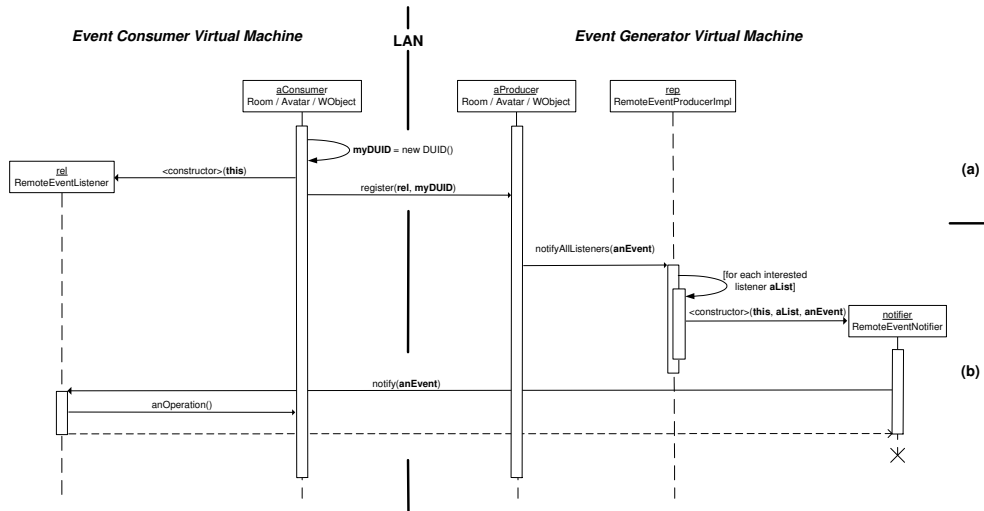


Fig. 6. Pattern used for integrating the event model in the framework



**Fig. 7.** (a) Setup of the event model (b) Notification of an event

Indeed, there are some utility classes, used by the framework, that allow rooms and avatars to register themselves to remote Jini lookup services and behave as normal Jini services.

But it is important to note that this capability does not replace the previous more classical manner, which consists in registering rooms and avatars into local rmregistries. The combination of this two publishing methods offers great advantages, without altering the scalability of the world.

In fact, the use of Jini lookup services offers great benefits for finding rooms and avatars when one have no *a priori* knowledge of the running world (i.e. on which machine to find a room for example). In addition to that, the template matching mechanism of Jini lookup presents a lot of possibilities (e.g. searching all german speaking avatars or all rooms intended for games). Unfortunately, these lookup services become bottlenecks if the world becomes too wide and counts a lot of room services they have too manage.

The classical rmregistry approach offers the shallow and flat "search by name" lookup method, but offers a perfect performance and avoids bottleneck then each rmregistry only contains the rooms and avatars running on one machine, and those are not so many (by the decentralize nature of our world).

So we hope we have the best of both worlds!

### 3.4 Some Examples of Objects

In various students projects, the MaDViWorld framework was intensively used and tested. So, there is already a little collection of interesting objects available. Here is a non exhaustive list of them:

- *Battleship*. This is a complete version of the legendary distracting two player board game. Its user interface includes graphics and sounds (see Figure 8).
- *Minesweeper*. Analog to the battleship, but it is a single user object.
- *Tamagotchi*. These little objects or virtual pets their owner has to take care of, present the peculiarity that they are affected by what happens to their siblings. For instance, if another tamagotchi in the same room dies, they feel sad and their own health level will decrease. That's why one could say that they are an example of "social" objects.
- *Whiteboard*. This simple version represent a basic collaborative editor.

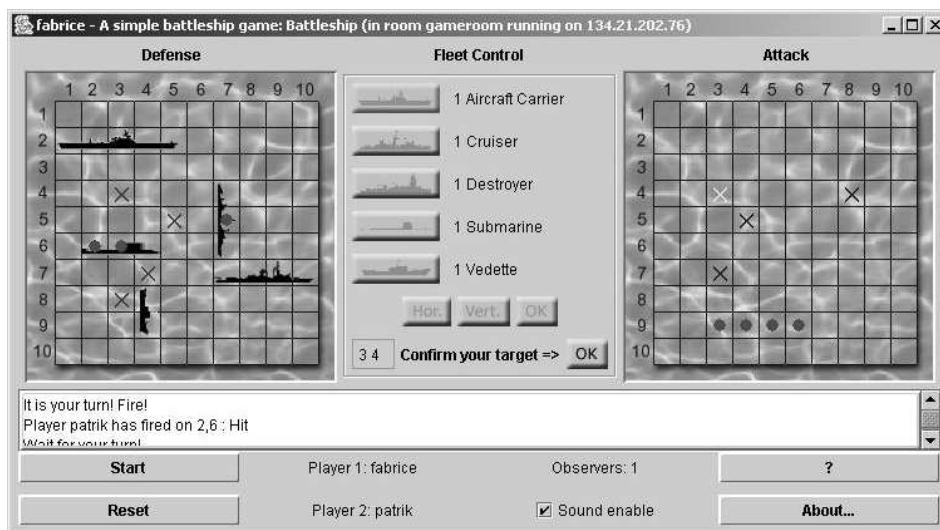


Fig. 8. A battleship game for MaDViWorld

## 4 Conclusion

### 4.1 Achievements

The actual version of the MaDViWorld framework has reached the following achievements:

- It is an *application framework* for highly distributed virtual worlds. It offers programmers the opportunity to transparently develop any kind of objects and test them in a virtual world, with all the advantages of mobility, remote execution, persistence, etc.

- As avatars, rooms and objects are distributed on several machines and are all interacting within the same world, an efficient *distributed event mechanism* is provided by the framework.
- In order to locate existing objects and services (room servers, rooms, avatars), the framework provides a *dual lookup mechanism*. On the one hand, it makes use of *rmiregistries* since its very first version. On the other hand, its latest release offers an important additional option. Indeed, it allows rooms and avatars to register themselves to remote Jini lookup services (*reggie*) and to act as well-behaved Jini services.
- Avatars can execute an object remotely, but the graphical user interface of the object always runs locally. As the avatars has no *a priori* knowledge of the kind of objects they will meet during their trip through the world, the framework provides a *graphical interface transport mechanism*. The core idea is that the graphical user interface and the implementation parts of an object are always clearly separated.
- The transport of *mobile objects* from one room to another is complex because it occurs in fact from one machine to another. So, when an avatar takes an object to carry it with her into a room running on a completely different machine, all the resources and class files relative to this object have to be moved along with it. This is all kept transparent for the end-user and hidden from the framework user.

## 4.2 Future work

The framework is still evolving and some enhancements will be integrated in the coming versions and will improve the efficiency and the conviviality of the various applications. Some efforts will also focus on the documentation and the refactoring (see [9]) of the framework for an even better understanding and readability for the user. But future research will cover the following five main points:

1. *Security*. On the one hand a security model inside the world (access to rooms, permissions,...) should be clearly defined and developed. On the other hand, security issues at the lower level of the framework should be resolved (downloading of classes, codebase verification, trusting of lookup services, etc.). An overview of existing solutions can be found in [?].
2. *Scripting*. The end-user should be able to change the behavior of objects at runtime.
3. *Static description*. Rooms and avatars with all their attributes should be described statically in a form of structured data such as XML and created "on the fly" by a parser tool. Persistence and state recovery mechanisms could then take advantage of this feature.
4. *Multimedia and 3D*. More complex space aspects of virtual worlds (rooms, avatars and objects) should be integrated.
5. *Real world example*. Developing concrete practical applications of the framework like educational worlds as virtual campus could be a possibility. Such a world is sketched in [6].

## References

1. Arnold, K. et al.: The Jini Specification, Addison-Wesley:Reading, MA, 1999.
2. Buschmann, F. et al.:Pattern-Oriented Software Architecture - A System of Patterns, John Wiley and Sons, 1996.
3. Deriggi, FV. et al.: CORBA platform as support for distributed virtual environments. Proceedings of the 9th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2001 (WSCG'2001), Plzen, Czech Republic, February 5-9, 2001.
4. Fabre, Y. et al.: A framework to dynamically manage distributed virtual environments. Virtual Worlds, Proceedings of the Second International Conference, VW 2000, Paris, France, 5-7 July. Springer: Berlin, 2000; 54-64.
5. Frécon E., Stenius M.: DIVE: A scaleable network architecture for distributed virtual environments (special issue on Distributed Virtual Environments). Distributed Systems Engineering Journal 1998; **5**(3):91-100.
6. Fuhrer, P., Kouadri Mostfaoui, G., Pasquier-Rocha, J.: MaDViWorld: a Software Framework for Massively Distributed Virtual Worlds. Software Practice and Experience, 2002, **32**:645-668.
7. Fuhrer, P., Kouadri Mostfaoui, G., Pasquier-Rocha, J.: The MaDViWorld Software Framework for Massively Distributed Virtual Worlds: Concepts, Examples and Implementation Solutions. Department of Informatics Internal Working Paper no 01-23, University of Fribourg (CH), Switzerland, July 2001.
8. Fowler, M., Scott, K.: UML Distilled: a brief guide to the standard object modeling language, 2nd Edition, Addison-Wesley, 2000.
9. Fowler, M.: Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2000.
10. Gamma, E. et al.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series: Reading, MA, 1995.
11. Greenhalgh, C., Benford, S.: MASSIVE: A distributed virtual reality system incorporating spatial trading. Proceedings 15th International Conference on Distributed Computing Systems. IEEE Computer Society Press: Vancouver, Canada, 1995, 27-34.
12. Gachet, A.: A Software Framework for Developing Distributed Cooperative Decision Support Systems - Construction Phase, Working Paper no 02-02, University of Fribourg (CH), Switzerland, March 2002.
13. Department of Informatics, University of Fribourg (CH), Software Engineering Group. *MaDViWorld: a Software Framework for Massively Distributed Virtual Worlds* <http://diuf.unifr.ch/softeng/projects/madviworld/> [2 December 2002].
14. Johnson, R. E., Foote B.: Designing Reusable Classes. Journal of Object-Oriented Programming **1**(2), June/July 1988, pp. 22-35.
15. Kirsh, D.: Adaptive rooms, virtual collaboration, and cognitive workflow. *Cooperative Buildings-Integration Information, Organization, and Architecture (Lecture Notes in Computer Science)*, Streitz N (eds.). Springer: Heidelberg, 1998;94-106.
16. Larman, C.: Applying UML and Patterns, Prentice Hall PTR:Upper Saddle River, NJ, 2002.
17. Li, S.: Professional Jini, Wrox Press Ltd.:Birmingham, 2000.
18. Pree, W.: Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1995.
19. Wilson, S., Sayers, H.; McNeill, MDJ: Using CORBA Middleware to support the development of distributed virtual environment applications. Proceedings IEEE Virtual Reality. IEEE Computer Society Press:Los Alamitos, CA, 1999, 8-13.