Mobile Agents In \textit{MaDViWorld}

Nicola David Fankhauser

November 2004

\textbf{Thesis Supervisors:}
Prof. Jacques Pasquier-Rocha
Dr. Patrik Fuhrer

Submitted to the Software Engineering Group, Department of Informatics
in partial fulfillment of the requirements for the degree of
Master of Science at the University of Fribourg, Switzerland.

Software Engineering Group
Department of Informatics
University of Fribourg, Switzerland
Abstract

This paper shows how mobile software agents can be implemented in the MaDViWorld (Massively Distributed Virtual Worlds) framework. After setting the stage by exploring the field of software agents, the MaDViWorld framework is presented and analyzed with the goal of implementing agents with it. A set of three agents are then presented in order to analyze the requirements to the underlying framework more accurately. These requirements are considered at both design and implementation level. The paper concludes with an analysis of the improved framework as a general agent platform with respect to the impact of the changes on its use as a massively distributed virtual worlds framework.
Acknowledgments

I would like to thank Patrik Fuhrer for his supervision and helpful, constructive advice, Prof. Jacques Pasquier-Rocha for making this thesis possible, Michael Schmehl for his clever algorithm, and Gaudenz Alder for his JGraph framework. A big thank you to all the people who supported me during the time of this master thesis, especially my family, my friends and everybody at mediagonal ag, couniq gmbh and nts ag.
As always, my thanks and love to Catherine.
### Contents

1 Introduction ........................................ 7
   1.1 Objectives ........................................... 7
   1.2 Time Line ........................................... 8

2 Mobile Software Agents ................................ 9
   2.1 Taxonomies ........................................... 9
   2.2 Intelligent Agents .................................. 9
   2.3 Pros and Cons of Mobile Software Agents ............ 10
   2.4 Frameworks, Toolkits and Libraries ................. 12
      2.4.1 Frameworks ..................................... 12
      2.4.2 Toolkits and Libraries ......................... 13

3 Overview of MaDViWorld .................................. 14
   3.1 Architecture ......................................... 15
   3.2 Objects in MaDViWorld ............................. 17

4 Agents in MaDViWorld ..................................... 19
   4.1 “Hopper” ................................................. 19
   4.2 “Explorer” .............................................. 20
      4.2.1 Discovering the Virtual World ................. 21
      4.2.2 Implementation .................................... 22
      4.2.3 Usage of the JGraph Framework ................. 24
      4.2.4 Automatic Graph Layout .......................... 26
      4.2.5 Visualization of the Graph ....................... 27
      4.2.6 Export as XML File ............................... 30
      4.2.7 Export as Plain Text File ....................... 30
   4.3 “Matchmaker” ......................................... 31
      4.3.1 Mode “Listen” .................................... 33
      4.3.2 Mode “Explore” ................................... 34
   4.4 Further Examples .................................... 35
      4.4.1 Distributed Audio Compression .................. 35
      4.4.2 Distributed Digital Still Images Decoding .... 36
5 Changes to MaDViWorld
5.1 Security and Agents in MaDViWorld ...................................... 38
5.2 Improved Code Mobility .......................................................... 41
5.3 Porting to Linux ....................................................................... 43
5.4 Console Version of XML Setup .................................................. 43
5.5 Versioning System .................................................................. 44
5.6 Build System .......................................................................... 45

6 Suitable Improvements
6.1 Dynamic Reloading ................................................................. 46
6.2 Persistence ................................................................................. 46
6.3 Programmable Timers .............................................................. 47
6.4 Centralized Directory ............................................................... 48
6.5 Communication ....................................................................... 48
6.6 Agent Communication Language ............................................ 49

7 Conclusion .................................................................................. 51

A Using Ant to Build and Run MaDViWorld ................................. 53
B MaDViWorld and Firewalls ......................................................... 54
C Linux and RMI ............................................................................. 55
D Testing on Multiple Machines ...................................................... 57
E Source Code and Related Material ............................................... 58
F Used Software ............................................................................. 59
G License ......................................................................................... 60

References ..................................................................................... 61
List of Figures

1 Agent taxonomy (according to [20]) ........................................ 10
2 SpamAssassin example rule checking ROT13-encoded email addresses .... 11
3 The tamagotchi object ............................................................ 15
4 Overview of the MaDViWorld framework .................................... 16
5 “Hopper” window with history .................................................. 19
6 “Explorer” window after exploring .......................................... 20
7 “Explorer” breadth-first algorithm in pseudo code ...................... 21
8 DTO classes and their relationships ........................................... 22
9 Visitable Interface ................................................................. 23
10 Visitor classes and their relationships ....................................... 24
11 Visitor Interface ................................................................. 25
12 How hooks are used in DTOs ................................................ 25
13 JGraph example ..................................................................... 26
14 Visualizing a graph in “Explorer” using JGraph ......................... 29
15 XML File Example ..................................................................... 30
16 Plain Text File Example .......................................................... 31
17 Screenshot of a “Matchmaker” agent ....................................... 32
18 Flow diagram of the “Matchmaker” agent ............................... 33
19 Example of email generated by a listening “Matchmaker” agent ...... 34
20 Using MaDViWorld to distribute load when converting audio data .... 36
21 Old security code excerpt ....................................................... 38
22 Challenge-response code excerpt ......................................... 39
23 Challenge Response scheme sequence diagram ....................... 40
24 Old code mobility scheme ...................................................... 41
25 New code mobility scheme ..................................................... 43
26 Old XML Syntax ..................................................................... 44
27 New XML Syntax ..................................................................... 44
28 Using underscores to mark stateful fields ................................ 47
29 Timer API proposal ............................................................... 47
30 Federated system with two federations ................................... 49
1 Introduction

The project MaDViWorld is a software framework implementing massive distributed virtual worlds. A virtual world consists of rooms connected by doors. Different types of objects (like board games, clocks etc.) can be placed in these rooms. Avatars (controlled by human users) can manipulate these objects, take them, wander to another room and put them down there.

Most implemented objects in MaDViWorld are passive. They cannot sense their environment, and consequently neither react to it. The implementation of a tamagotchi is an exception: It gets “sad” if another tamagotchi in the same room dies. This behavior means that the tamagotchi object is reactive.

However, active objects are missing in MaDViWorld. An active object could “decide”, based on the sensation of their environment (the room it is in, other objects in the same room etc.) what action to perform (move to another room, communicate with other objects etc.). This type of object belongs to the class of mobile software agents.

The main objective of this paper is to investigate whether it is possible to implement mobile software agents using the MaDViWorld framework.

This paper is laid out as follows: After the introduction and the declaration of objectives, a theoretical overview of Mobile Software Agents is given, a selection of some specialized agent frameworks presented, followed by a short technical overview of the MaDViWorld framework (Chapters 1, 2, 3). The second part (Chapter 4) details in both theory and practice the realization of three mobile software agents on the basis of MaDViWorld. The next part (Chapters 5 and 6) discusses what changes were made and which changes would be favorable to the MaDViWorld framework. Finally, the last Chapter attempts to draw a conclusion.

1.1 Objectives

The objectives of this paper are the following:

1. Study and understand the architecture of MaDViWorld;
2. Position the framework with respect to existing agent platforms;
3. Present the current research on mobile software agents;
4. Present existing dedicated agent platforms;
5. Determine whether MaDViWorld can be adapted to serve as agent platform and perform these changes;
6. Implement two agents based on MaDViWorld: First, an agent called “Explorer” that draws a map by visiting all rooms in a given virtual world. Second, an agent called “Matchmaker” that fixes meetings with other agents of the same type on behalf of their respective owners.
1.2 Time Line

**August 2003**  Start of the work, familiarizing with the **MaDViWorld** project. Reading papers by Patrik Fuhrer et al. ([23], [24]). Meeting with Prof. Pasquier and Patrik Fuhrer. Adding first **Ant** scripts.

**December 2003**  Solving problems with porting to Linux, Jini and **RMI**. Reading books ([49], [5]). Refactorings and code cleanups.

**January 2004**  **CVS** installation, commit scripts adapted. First draft of this paper. Meeting with Prof. Pasquier and Patrik Fuhrer.

**February 2004**  Tests with multiple machines (running Linux). **XML** setup tests and adaptations. Hopper agent written.

**March 2004**  Ideas for distributed computing agent systems. First version of Explorer agent in **CVS**.

**April 2004**  New version of Explorer agent. Implementation of physical graph lay outer.

**May - June 2004**  Major progress with the paper, theoretical issues. Tests and analysis of other agent platforms and frameworks.

**July - August 2004**  Implementation and testing of the matchmaker agent.

**September - November 2004**  Final works on the paper, solving problems with dead-locks, refinement of the exploring algorithm.
2 Mobile Software Agents

In the past, a lot of different definitions emerged of what software agents are. For some, common programs already qualify as agents, while for others more stringent conditions like artificial intelligence, communication and collaboration with other agents must be met.

However, there are certain desired properties characterizing software agents that all these definitions have in common (taken from [20]):

- reactive: responds in a timely fashion to changes in the environment;
- autonomous: exercises control over its own actions;
- goal-oriented: does not simply act in response to the environment;
- temporally continuous: is a continuously running process;
- communicative: communicates with other agents, perhaps including people;
- learning: changes its behavior based on its previous experience;
- mobile: able to transport itself from one machine to another;
- flexible: actions are not scripted;
- character: believable “personality” and emotional state.

2.1 Taxonomies

According to [20], an autonomous agent can be defined as:

An **autonomous agent** is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.

Because there exist so many definitions of what agents are, should do and should not do, a unifying taxonomy like in Figure 1 seems more appropriate than to delve into subtle differences between those definitions. It is a very broad approach, and thus includes biological and robotic agents as well.

2.2 Intelligent Agents

For quite a long time, research on artificial intelligence was believed to result eventually in human-like computer programs. It was hoped that they will exhibit a kind of human intelligence, yet would not make mistakes like humans do. When discovering that computers were good at things human were not so good at, this lead to the false assumption that it was only a matter of time teaching a computer to do things
Humans are good at. However, these goals could not be achieved until today. Expectations seemed to be too high, or at least pointed in the wrong direction.

It seems to make more sense to provide software agents with a simpler form of intelligence. An intelligent agent designed to collect mail from different hosts and gathering it at one central place may exhibit very strange behavior when it suddenly cannot connect to its central host anymore, because it could decide to search another (potentially insecure) place to store the gathered mail. This small example only tries to show that every form of augmented intelligence for one task implies additional intelligent behavior to cope with potentially wrong deductions or actions.

Inspired by this I tried to implement agents that help a human user doing complex tasks, but expose intelligence that is still understandable by the user. A human user needs to have a simple working model of the inner workings of the agent so it can be used efficiently.

2.3 Pros and Cons of Mobile Software Agents

Mobile software agents are suitable where the relative complexity of task is clearly higher compared to the users own resources: Microsoft Word’s *clippy* is a good example of an agent helping users not wanting or being able to read the manuals and using the online help. With *clippys* assistance, one can write letters on a computer with almost no knowledge about how word processing works. *Clippy* can be regarded as agent, as it fulfills all characteristics discussed at the beginning of this Chapter\(^1\).

However, this agent may become annoying for users wanting to do more complex (or just other) things

\(^1\) *Clippy* is not mobile, it qualifies nevertheless as agent (albeit not as mobile agent).
compared to what *clippy* (with its limited intelligence) thinks she wants to do. This is the point where
the agent is superfluous and thus does more harm than good\(^2\).

SpamAssassin [56, 53] is another agent, albeit not mobile, that a lot of people use (most of them unknow-
ingly). The purpose of this agent is to filter out unsolicited commercial email\(^3\). It does so by applying
rules to incoming messages. For each rule that matches, a certain score is added internally. If the total
score is above a certain limit, the email is tagged as being probably spam. There are static rules, like the
example shown in Figure 2\(^4\).

| body      | EMAIL_ROT13  /\b[a-z]([-]+\^[a-z-]+\{(a-z){2,3}\}\b/ |
| describe  | EMAIL_ROT13 Body contains a ROT13-encoded email address |
| test      | EMAIL_ROT13 ok qhabs`ebtre(pbz |
| test      | EMAIL_ROT13 ok zxrggyre`riv-vap(pbz |
| test      | EMAIL_ROT13 fail duncf-nospam@rogers.com |
| score     | EMAIL_ROT13 4.400 |

**Figure 2:** SpamAssassin example rule checking ROT13-encoded email addresses

But because handmade rules age quickly\(^5\), SpamAssassin incorporates a learning algorithm based on
Bayes’ theorem\(^6\). This allows training of SpamAssassin by feeding it with misclassified messages.

Text classification does not necessarily need restrict itself to two classes (like spam and non-spam mes-
sages), but could be used in a mail client application to move new messages to the correct folder. For
example, mails from mailing list A would be moved to folder MailingLists/A. New folders would need
to be trained by and by. The alternative (and more common) way to achieve the same effect is to define
handmade filters. Obviously, these filters always catch the correct messages and don’t make mistakes.
However, these filters are not very flexible towards unknown material and easily fail under such circum-
stances.

In contrast, heuristic algorithms (like SpamAssassin’s Bayes learning algorithm) can handle previously
untrained and new material successfully, but always with a certain level of incertitude\(^7\). They produce
false-negatives (spam not recognized) and false-positives (solicited mail erroneously classified as spam).
The latter is the least wanted, as can have more serious consequences compared to spam messages not
detected.

It is not possible (or advisable) either to simply delete messages classified automatically as spam, for the
\(^2\)This is the point where most users disable it fully.
\(^3\)Also called spam, hence the name of SpamAssassin.
\(^4\)The ROT13 encoding scheme is a Caesar cipher with a shift of 13: A becomes N, B becomes O, and so forth, down to Z, which
becomes M. Because there are 26 characters in the alphabet, the same shift is used to decode a ROT13-encoded message. See
[66] for more details.
\(^5\)Static rules can easily be circumvented by spam sending people, as those rules are known publicly, another advent of open
source.
\(^6\)See [31] and [32] for more information.
\(^7\)This applies to known and trained material as well, though to a lesser extent.
reasons discussed above. This means a human has to scan discarded messages regularly to prevent that a perfectly legal message gets deleted.

These two points show that even very successful agents like SpamAssassin have their limits, as there are fields of application where other tools are more appropriate. Generally, the limits seem to be that the artificial intelligence practically possible today is too restrained. Another problem appears to be the imprecision of the applied heuristic algorithms (with their inherent error rates).

2.4 Frameworks, Toolkits and Libraries

This section will cover in short existing frameworks, toolkits and libraries which alleviate the design, implementation and deployment of mobile software agents.

2.4.1 Frameworks

**Jacomma** Java Communicating Agents. On its website [38], Jacomma is described as “an agent development platform/framework for developing distributed, mobile, and reactive information agents with heterogeneous communication capabilities, in Java and Jython[^8]. Jacomma provides a development framework and an execution environment, which is located on top of the “Inter-Agent Communication Model infrastructure” (ICM). Unfortunately, its development seems to have stopped in 2001. Because of this, it is based on an old ICM version not available anymore. However, having a well-designed architecture and because of its ease of use, it is still worth mentioning.

**JADE** Java Agent DEvelopment Framework, complies with FIPA[^18] specification (middleware and ACL[^9]). It is “a software framework fully implemented in Java language. It simplifies the implementation of multi-agent systems through a middleware that complies with the FIPA specifications and through a set of graphical tools that supports the debugging and deployment phases” [39].

**JATLite** Java Agent Template Lite, uses KQML ACL[^12], includes an Agent Message Router (AMR). It allows robust communication between agents on the Internet [44].

**JiVE** JAFMAS (The Java Agent Framework) integrated Visual Environment. It is “a tool for the design and development of multi-agent systems. JiVE allows a designer to draw a multi-agent system graphically (including communications and behavior), specify the necessary properties, check the design for correctness, and deploy their application either directly in the JAFMAS framework or in a framework of their own choosing.” [47].

[^8]: Formerly called JPython, which has been renamed and superseded by Jython [48].
[^9]: Agent Communication Language
2.4.2 Toolkits and Libraries

**JFIPA** FIPA XML-based Message support for Software Agents. It is “a set of Java-based tools that supports parsing and routing FIPA Agent Communication Language (ACL) messages represented in XML.” [62].

**ICM** Inter-Agent Communication Model. It is “a communication mechanism that can be used for sending messages between agents in an asynchronous fashion. Its intended application area is as a transportation mechanism for agent communication languages (ACLs), such as KQML and FIPA’s ACL.” [36].
3 Overview of MaDViWorld

The MaDViWorld project was started in 2001 by Patrik Fuhrer within the scope of his doctorate at the University of Fribourg, Switzerland. The acronym MaDViWorld stands for Massively Distributed Virtual Worlds. It is a software framework written entirely in Java [41] which aims to implement a software adhering to the virtual world paradigm [24, p. 3]. A Virtual World has typically these five properties [55, cited by [24]]:

- A shared sense of space: All participants have the illusion of being in the same room or space in general.
- A shared sense of presence: When a new participant enters the shared space, all other participants are notified in some form of its arrival. Likewise, if somebody leaves the shared space, the remaining participants are notified of its departure as well.
- A shared sense of time: Participants should be able to see events generated by them, others or objects as they occur, i.e. in real time.
- A way to communicate: Participants can communicate with each other. This can be either text, voice, video or any other form of communication.
- A way to share: Participants can interact with the virtual environment. They can inspect, touch, manipulate and even take and put down objects.

MaDViWorld addresses all these points and thus qualifies as a Virtual World platform:

- A shared sense of space and presence: Users of MaDViWorld (called avatars) enter a virtual room. A list of avatars and objects is displayed, and all avatars are notified if other avatars enter or leave the room.
- A shared sense of time: Apart from network delays, actions of avatars and objects are visible in real time.
- A way to communicate: Older versions of MaDViWorld had a chat panel built in directly into the client software. Newer versions make use of special chat objects that can be used to communicate.
- A way to share: Avatars can inspect, manipulate and take or put down objects.

During its 3 year development stage, MaDViWorld has matured and undergone various re-design and improvement cycles. Features like security that were not a primary goal to achieve in the first place are addressed at the time of writing.

Numerous student projects added working implementations of objects to the MaDViWorld framework. Collaborative games, a clock, a music player, a chat and a white board are amongst the most advanced
objects. As noted above, instances of these objects can be taken by an avatar from one room and put down in another.

One of the most advanced objects however is the tamagotchi (see Figure 3), developed by A. Invernizzi within the scope of his diploma thesis [37]. It is a simulation of a pet that needs to be taken care of by an avatar. If not, it will eventually die. The current implementation senses if a tamagotchi in the same room dies and gets ‘sad’ itself. Thus, a tamagotchi is a reactive object.

However, there were no active objects in MaDViWorld up to now. The goal of this paper is to analyze if and how active objects can be implemented in MaDViWorld. Commonly, active objects are called Software Agents, because they are ultimately the agency of a human user, i.e. take actions on the user’s behalf. Given the distributed nature of the framework, an important feature is the autonomous movement of such an active object, leading to Mobile Software Agents.

3.1 Architecture

MaDViWorld is a framework for the creation of massively distributed virtual worlds. Figure 4 shows an overview of the framework. The core package forms the base of the framework and consists of the most important interfaces such as core.Room, core.Avatar and core.WObject. With the exception of core.WObject, these interfaces have a default implementation (packages core.avatar and core.room respectively, both marked in Grey). These interfaces (including core.WObject) can be overridden and extended with a specific implementation (marked in Orange). While normally only one implementa-
tion of core.Room and core.Avatar will be used, there are a lot of different objects implementing the
core.WObject interface, providing simple services like displaying the current time or more complex
objects like board games.

Figure 4: Overview of the MaDViWorld framework

The util package contains various sub-packages and classes that are used throughout the framework. The
event package finally is comprised of classes implementing the framework’s event mechanism.

The key components of MaDViWorld according to [26] are:

Avatar: The human users need representations in the virtual world and are therefore personified by an
avatar. Through her avatar a user can walk, look around the virtual world, manipulate objects
and perform virtual actions. In other words, avatars allow users to interact with the virtual world
and other users and also allow navigation through the world. We see one’s avatar as being your
representative in Cyberspace. In text-based virtual realities, such as MUDs, one’s avatar consists
of a short description which is displayed to other users who have their avatars ‘look’ at you. In
a 3D graphical world, the avatar can take the shape of an animated cartoon or of your favorite
fantasy hero.

Object: In the discussed virtual world, there are objects, not just simple passive data objects, but active
objects the avatar can execute (e.g. the Fibonacci calculator). These objects can even be multi-user
(e.g. the two players “tic-tac-toe” board) and can have multiple observing avatars. Last, but not
least, objects can be copied and/or moved by the avatar.
Location: Avatars and objects have a location. This concept is natural and necessary, since it supports navigation.

Navigation: The action of going from a given location to another. Avatars can do that, either if there is a link between these two locations, or directly if they know the address of the subspace.

Subspaces: Each location can be seen as a subspace of the whole virtual world. It is natural to consider that the shared virtual space is composed of many different subspaces. Furthermore, the avatars and objects are always contained within one given subspace.

Event: The avatar has to be aware of its environment. This awareness is achieved by the concept of events. An avatar moving, a played move in a “tic-tac-toe” game are simple examples of such events.

Event producer: An event always has a source which produces it. For instance, a game could produce a “game finished event”.

Event consumer: An event can be caught and interpreted by an event consumer, which then reacts properly or simply ignores it. For instance, the player of the game understands that the game is finished.

Event propagation: Each event has an event propagation space. This space is a delimited zone around an event producer, within which an event consumer will be aware that the event has occurred.

3.2 Objects in MaDViWorld

This section discusses what common MaDViWorld objects are and how they compare to agents. According to [25], these objects already classify as agents. The lack of middle-ware services specific to agents however make the framework difficult to use as full blown agent platform. Not all types of agents need such services however; e.g. non-collaborative agents do not require a central directory or agent message routers.

In the current framework, objects have a persistent state\(^{10}\). They can be moved from one room to another: an avatar takes the object into her bag, then changes room itself and finally takes out the object from the bag and deposes the object in the room.

The objects’ business logic can be run remotely (i.e. in the current room) or locally, while the GUI\(^{11}\) always runs on the avatars machine. The difference between running an object remotely or locally can be easily demonstrated with the Clock object: If the time on the remote and local machine are different the clock object will show a different time, depending on whether it was run locally or remotely.

\(^{10}\)Objects can persist their state, but are not forced to. Simpler objects often do not need a persistent state.

\(^{11}\)Graphical User Interface
There are existing implementations of objects that collaborate (like fighting robots), others behave socially (like virtual pets\textsuperscript{12}).

Let us compare the capabilities of these objects with what we defined as characteristics of mobile software agents in Chapter 2:

- reactive: Objects are reactive towards the environment (e.g. a tamagotchi senses the death of another one in the same room).

- autonomous: An object does exercise control over its own actions.

- goal-oriented: Most objects are not really goal oriented, but react mostly only to user actions.

- temporally continuous: Objects can react to events from the environment, are thus temporally continuous.

- communicative: Objects can communicate with other objects through events. However, this is not standardized and needs additional logic to be implemented in the objects.

- learning: Objects can learn if suitable algorithms are implemented. It is not supported actively by the \texttt{MaDViWorld} framework.

- mobile: Objects were not thought to be moving autonomously. But because of the nature of the framework, it still can be done.

- flexible: Actions of an agent should not be scripted. This characteristic depends again heavily on the logic of the object. It renders objects complex, but it is double.

- character: Again, this depends on how complex an object is. Objects like tamagotchis have character, while others like the clock have not.

Summarizing, \texttt{MaDViWorld} objects can be regarded as agents, if we restrain ourselves to only some important characteristics like reactivity, autonomy and temporal continuity. Other characteristics\textsuperscript{13} can be built into \texttt{MaDViWorld} objects, but require more implementation work, and yield in more complex, less supportable agents, because the framework does not provide the required services.

---

\textsuperscript{12}The mentioned object implements a virtual tamagotchi.

\textsuperscript{13}Attributes like goal-oriented, communicative, learning and flexible.
4 Agents in MaDViWorld

When the framework was invented, the original idea was to have a massively distributed system for virtual worlds, having rooms, avatars and objects. It was not meant to become an agent platform. As such, the lack of special services make it unsuitable to implement general agents. The framework already provides essential services like strong code mobility, communication and the surrounding infrastructure (like setup and client applications) of a distributed environment. Because MaDViWorld resembles a lot the real world, the agents sensing of its environment is eased. Also the testing of agents behavior in a semi-real world with avatars (i.e. people), rooms and objects proves itself to be easier.

There are however certain fields of application where agents can be useful for an environment like MaDViWorld. The following sections show three implementations of such agents, as well as two drafts of more complex agent systems that have a less strong reference to virtual worlds.

4.1 “Hopper”

![Figure 5: “Hopper” window with history](image)

This agent can move itself from one room to another (also across machine boundaries). It has a simple GUI where the user can enter the destination host and room name. Upon pressing the button Move, the agent closes the GUI, copies itself to the destination room on the destination server and removes itself from the current room.

Figure 5 shows the agent after having been in room R1 and R2 on host 192.168.0.205 and in room R7 on host 192.168.0.127. A text field is filled with the entire history of the agent and includes host name,

14The framework does not copy the stack of the object but rather saves the state field and copies it back upon the completion of the movement.
room name and which objects and avatars being present (type, name and description of each) at the time of visit.

This agent is not autonomous and acts on orders of the human user, although the agent could as well be controlled by a program.

4.2 “Explorer”

The “Explorer” agent allows to discover an existing MadViWorld world. It is a normal MadViWorld object in the sense that it has to be installed, run, taken etc. just like ordinary objects. It differs in only one aspect: it is capable to move autonomously from room to room and from host to host.

The mobile agent queries the following properties and entities of any visited room:

- Room name, host name;
- Objects in the room, their type, name and description;
- Avatars present in the room, their type, name and descriptions;
- Doors of the room, to which rooms on which hosts they lead.

In addition to MadViWorld intrinsic information an extended implementation could try to gather different information about each machine in the network. These additional features are not present in the current implementation of the agent:

- Role of the machine;
- Number of users logged in;
- CPU load and speed;
- Memory size and current usage;
- Fixed memory size and current usage;
- Network speed and delay to its neighbors.

The topology of the virtual world is visualized with a graph. The graph contains all hosts, rooms, their avatars, objects and doors to other rooms. Hosts, rooms, avatars and objects are shown as vertices, whereas doors and the relations “room belongs to host”, “object is in room” and “avatar is in room” are drawn as edges between the corresponding vertices.

### 4.2.1 Discovering the Virtual World

The “Explorer” agent wanders around the world like a human-controlled avatar would do. It discovers new rooms by following doors from other rooms. The algorithm is a breadth-first search algorithm, as opposed to depth-first search algorithms. Figure 7 shows the exact steps in pseudo code.

1. Get the current room and host.
2. Inspect the current room (objects, avatars and doors).
3. For each door, look if the target host/room pair of the door is already in the internal graph.
   - If no, add the target host/room pair to the graph.
4. Look if there is a room on the current host not visited yet.
   - If yes, move the agent to this room. Go to step 1.
   - If no, look if there is a room not visited yet on a host in the graph.
     - If yes, move the agent to this room. Go to step 1.
     - If no, move the agent back to the starting room and stop.

**Figure 7:** “Explorer” breadth-first algorithm in pseudo code

In order to faithfully reproduce biological behavior, the agent can only discover rooms and hosts if they are connected directly or indirectly by rooms with the starting room\(^\text{15}\). This means that, depending on where the agent is started, a different view of the same world may be produced\(^\text{16}\).

The current implementation of the framework would even permit to query almost all needed information\(^\text{17}\) from one single host over the network. This would have been possible without the use of a mobile software agent, and was therefore of no interest.

---

\(^{15}\)In graph theory, the condition is that if in a given graph \(G = (V, E, I)\) two vertices \(i, j \in V\) are connected by a path, an agent starting in vertex \(i\) will eventually discover vertex \(j\).

\(^{16}\)This will be the case if agent \(a_1\) starts at vertex \(i\), agent \(a_2\) at vertex \(j\) and there is no path between vertices \(i\) and \(j\).

\(^{17}\)With the exception of machine-dependent information like CPU load, memory size etc. i.e. data not provided by the framework itself.
Because the same host can be identified by multiple IP\textsuperscript{18} addresses and DNS\textsuperscript{19} entries, care had to be taken when implementing so that the agent would not visit the same host more than once. To identify a host uniquely, the MaDViWorld framework was changed to supply each host with a DUID\textsuperscript{20}. With this new feature, the “Explorer” agent is able to map doors with different destination host names (yet pointing to the same logical host) to the same internal representation. Consequently, no double (or even more) entries of the same host in the internal structures are possible.

### 4.2.2 Implementation

To allow simple manipulation of the gathered data of hosts, rooms, doors, objects and avatars, the structure is mapped into a tree. Each object (host, room, door, object or avatar) is represented by a DTO\textsuperscript{21} node. See Figure 8 for an UML diagram of all DTO classes and their relationships.

![DTO classes and their relationships](image)

Figure 8: DTO classes and their relationships

---

\textsuperscript{18}Internet Protocol, see RFC 791 \[11\].

\textsuperscript{19}Domain Name System

\textsuperscript{20}Distributed Unique Identifier, see also \[23\].

\textsuperscript{21}Data Transfer Object \[57\]
Each of these DTOs implements the Visitable interface, shown in Figures 9 and 10. A visitor is a software design pattern described in [28]. It allows classes implementing a common interface (here called Visitor) to traverse tree-like structures (implementing a common interface, here called Visitable) in a pre-defined manner.

```java
public interface Visitable {
    public void visit(Visitor visitor);
}
```

**Figure 9: Visitable Interface**

All classes implementing the interface are able to accept an object of type Visitor with the method visit(). A visitor must implement the Visitor interface, shown in Figure 11. It consists of the different callback() methods, for each DTO type one method. Upon receiving a visitor with the visit() method, the class calls back the visitor immediately by passing itself to the visitor method callback() like this: visitor.callback(this). The visitor, depending on the actual implementation, may now extract information from the received object. After that it exits the method, thereby giving control back to the original DTO. The DTO now passes the visitor object to all its children objects by calling their respective visit() method.

Before calling back the visitor, each DTO calls a special begin method named beginHook{TypeOfDTO}DTO(), and before exiting the visit() method, endHook{TypeOfDTO}DTO() is called. This allows the visitor to know what kind of DTO it will get with the callback() method.

If a DTO sends the visitor to a list of child objects, it calls a special method as well - one before calling any child, named beginHook{TypeOfChildDTO}ListDTO(), and one after the call loop, named endHook{TypeOfChildDTO}ListDTO(). This is done to indicate what kind of child objects the visitor has to expect. Mainly, these hooks have been introduced to make the XML export possible. See Figure 12 for the code snippet.

An issue had to be addressed concerning state preservation. When the agent moves, its state is serialized by the framework and after successful arrival on the destination host, it is set back with the method setState(WObject otherObject). While developing the agent, it became apparent that while “lightweight” usage of this feature, e.g. with a Hashtable, worked like expected, but failed when the internal state consisted of a (complex) tree of objects, in this case a GraphDTO with its many children objects.

An admittedly not very performing, but working solution for this problem was the introduction of a pair of serializer/deserializer utility method in the class AgentUtil called serializeDTO(DTO dto) and deserializeDTO(String serialized). They use Sun’s sun.misc.BASE64Encoder() encoder and decoder to transform an object into a ASCII string and back into an object.

This functionality can be easily turned off by setting the field ExplorerImpl.useSerializer to true. It

---

22The begin hook creates the opening XML tag, the end hook the closing tag.
remains to be determined what component of the framework (or underlying layer) is responsible for this behavior.

### 4.2.3 Usage of the JGraph Framework

To achieve an efficient, robust and feature rich implementation the JGraph [45] framework, written by G. Alder, was used. This framework is a specialized Java Swing [40] component and as such easily integrated into existing Swing GUIs, like the MaDViWorld object GUI framework.

However, JGraph has its own kind of terminology and implementation specifics. In JGraph, an edge is not simply linked directly with two vertices; each vertex has zero or more so called ports, where edges are attached to. Furthermore, each object (be it an edge, a vertex or a port) has a list of attributes. Some specify the background color, others how the border should look like. These lists of attributes are called maps. To set these attributes, the JGraph framework provides the static class GraphConstants. According to the JGraph tutorial [2], it should be used like in Figure 13. Lines 6 - 10 show how to use the GraphConstants class. For a JGraph novice it is not very clear why an external class is used to manipulate attributes of a cell. Furthermore, this reminiscence of functional programming style does not
public interface Visitor {
    void callback(AvatarDTO visited);
    (...)  
    void callback(GraphDTO graphDTO);

    void beginHookAvatarDTO();
    void endHookAvatarDTO();
    (...)  
    void beginHookGraphDTO();
    void endHookGraphDTO();

    void beginHookAvatarList();
    void endHookAvatarList();
    (...)  
    void beginHookObjectList();
    void endHookObjectList();
}

Figure 11: Visitor Interface

fit very well into the object oriented nature of the framework as a whole: It relies on static procedures with
side effects (in this case the map object is changed during the call of a method of the GraphConstants
class) and separates logic (DefaultCell, DefaultPort etc. versus GraphConstants). Both points are hard
to understand for JGraph novices.

To ease the usage of the JGraph framework and allow simple and fast manipulation of the graph, especially
the addition of vertices and edges, a wrapper class named JGraphUtil was written. Essentially, it
contains these methods:

- **connectNodes(Node firstNode, Node secondNode)**

Figure 12: How hooks are used in DTOs
Excerpt from explorer.gui.JGraphUtil.insertNode()

```java
DefaultCell cell = new DefaultCell();

Map attributeMap = new Hashtable();

Map map = GraphConstants.createMap();
GraphConstants.setBounds(map, rect);
GraphConstants.setBorderColor(map, Color.black);
GraphConstants.setBackground(map, background);
GraphConstants.setOpaque(map, true);
GraphConstants.setBorder(map,
    BorderFactory.createRaisedBevelBorder());
attributeMap.put(cell, map);
DefaultPort port = new DefaultPort("Floating");
cell.add(port);
Object[] insert = new Object[] { cell };
model.insert(insert, attributeMap, null, null, null);
```

Figure 13: JGraph example

- `insertRoomNode(RoomDTO obj)`
- `insertObjectNode(ObjectDTO obj)`
- `insertAvatarNode(AvatarDTO obj)`
- `insertHostNode(HostDTO obj)`
- `convertGraph()`

The last method, `convertGraph()`, converts the internal representation of the virtual world topology (a tree with nodes) into different vertices and edges and adds them to the JGraph graph.

### 4.2.4 Automatic Graph Layout

The JGraph framework does not position vertices automatically so that they form a pleasant, human readable graphic. Even though a spin off product called JGraphpad does include different algorithms that automatically layout graphs, it was deemed to dull to port one of these complex algorithms. Instead, a more elegant solution was searched and found with GraphOpt [52], written by Michael Schmuhl. This software, written in C++, is an implementation of a fairly simple, yet elegant algorithm based on physics. With the consent of the author, I ported the C++ classes to Java.
Let me briefly describe the algorithm applied. Each node has a mass and is electrically charged. Because of the point charge, any two nodes repel. The amount of this force depends on their distance and their charge.

The electrical force between two point charges is defined as

\[ F = \frac{1}{4\pi \varepsilon_0 \varepsilon_r} \cdot \frac{Q_1 Q_2}{r^2}, \]  

where \( r \) is the distance between the two point charges \( Q_1 \) and \( Q_2 \), \( \varepsilon_0 \) is the Coulomb constant (\( \approx 8.85 \cdot 10^{-12} \text{CV}^{-1}\text{m}^{-1} \)). We assume \( \varepsilon_r = 1 \) (approximately air at normal pressure) and \( Q_1 = Q_2 \).

Edges are modeled as mechanical springs. The force of the pull exerted by a mechanical spring depends on its spring constant \( D \) and the displacement from ideal length in meters \((l - l_0)\):

\[ F_s = D(l - l_0) \]  

The forces of a spring pulls the two connecting vertices together until either the ideal length of the spring \( l_0 \) is reached or the repellent forces of the two vertices are in balance with the force of the spring. Unconnected vertices move away from each other until the electrical force is negligible (the electrical force diminishes in the square in respect of the distance).

As each iteration of the algorithm is independent, time is constant, thus \( t = 1 \). Therefore velocity equals distance: \( v = \frac{s}{t} = s \), leading to acceleration \( a = \frac{v}{t} = v = s \). As the force \( F = m \cdot a \), in this special case \( F = m \cdot s \), thus leading to \( s = \frac{F}{m} \).

The vectors of directed forces are split up into forces for the \( x \) and \( y \)-axis. Each vertex is assigned the sum of all forces applying to it. Finally, after all forces applying to all vertices have been calculated, each vertex is moved according to the force applied to it.

### 4.2.5 Visualization of the Graph

Once the topology of the virtual world has been discovered the agent returns to the room it was sent out from. The human user can start again the agent’s graphical user interface and the graph will be showed as connected vertices (as shown in Figure 14). Every vertex has additional information which can be displayed when letting the mouse rest over a vertex for more than two seconds. After this delay, a small rectangular area with textual information on it shows up\(^{23}\).

The user can actively move around vertices by clicking on a vertex and dragging it to some other location in the panel. Upon the release of the mouse button, the automatic physics lay-outer is activated and the graph gets reordered again. This can be done multiple times and allows to display the graph as wished by the user.

\(^{23}\)A “tool-tip.”
The physical lay-out algorithm is so appealing because of its simplicity, the resulting lay-out graphically separates clusters of vertices (rooms, doors, objects and avatars) that are highly interconnected from other, less connected vertices. This can be seen in Figure 14.
Figure 14: Visualizing a graph in "Explorer" using JGraph.
4.2.6 Export as XML File

If the user wants to save the current discovered topology the “Explorer” agent can save the graph as a XML file adhering to the world.dtd of the MaDViWorld student project XML Setup [50] (see also section 5.4). For saving the XML file, press the button “Save as XML” and specify a location in the popped up file browser and press “Save”.

The resulting XML file can be loaded by the XML Setup application and thus allows to re-create the saved topology at a later point\(^\text{24}\).

An example of the resulting XML file is shown in Figure 15.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE world SYSTEM
"http://diuf.unifr.ch/softeng/projects/madviworld/DTD/world.dtd">
<world>
  <worldname>world_127.0.0.1/corridor</worldname>
  <room>
    <roomname>corridor</roomname>
    <access>N/A</access>
    <owner>N/A</owner>
    <kind>N/A</kind>
    <host>127.0.0.1</host>
    <doorlist></doorlist>
  </room>
  <objectlist>
    <object>
      <objectname>myobject</objectname>
      <description>An exploring agent.</description>
      <type>explorer</type>
    </object>
  </objectlist>
</world>
```

*Figure 15: XML File Example*

4.2.7 Export as Plain Text File

The agent can as well save the current graph as a plain text file. Press the button “Save as plain text” to do this. This can look like shown in Figure 16.

\(^{24}\)However, this feature should not be regarded as a persistence mechanism. It does not preserve state, nor does it save the byte-code of encountered unknown objects.
4.3 “Matchmaker”

To show how agents can communicate by means of existing infrastructure in MaDViWorld, the matchmaker agent was conceived. Its aim is to let an agent fix meetings automatically on behalf of a human user. A matchmaker agent operates in one of these two modes:

**Explore mode** The agent actively roams the virtual world in search of other matchmakers in listen mode\(^25\).

**Listen mode** The agent stays in one room and waits for passing exploring matchmakers to send him a list of meeting propositions.

The agent needs to be configured first by filling in an email-address in the Email Address field. Additionally, the agent needs to know how to send emails, thus a configuration string must be entered in the **Mail Configuration** field\(^26\). The syntax is as follows\(^27\):

\[(username:password|username)@)?host\]

This means all the following configuration strings are valid:

- host

---

\(^25\)The roaming algorithm is the same as used in the “Explorer” agent. See Section 4.2.1 for a detailed discussion.

\(^26\)The agent uses **SMTP** (Simple Mail Transfer Protocol) to send emails.

\(^27\)This is a regular expression, see [21] for more information.
If a user wants to find possible meeting times with other users, she puts a matchmaker agent in a room, starts its GUI (see Figure 17 for a screen shot) and adds a set of possible time periods to the list of Own Dates. To further describe what kind of activity she would like to engage with (playing a board game, chatting etc.), she fills in the Description text area. After that, she clicks on the Explore! button. Her agent is put in explorer mode and begins to search all rooms it encounters for other matchmaker agents that are in listen mode.

Once the exploring agent finds such a matchmaker agent in listen mode, it sends him his list of dates, along with the description filled in by its user. The listening agent compares the list of received dates with his own list and returns a new list of matching time periods. Both agents now display this list in the Confirmed Dates list. Each entry of this list consists of the from/to dates, the respective names of the listener and exploring agents, their descriptions (taken from the Description text areas) and the email addresses of the two parties.

In the case of multiple matching time periods, the possible time periods could be prioritized by the user. This was not implemented however.
4.3.1 Mode “Listen”

Figure 18: Flow diagram of the “Matchmaker” agent

Figure 18 shows the flow diagram of the matchmaker in the two possible modes. In listen mode, the agent behaves as follows:

**Start** The agent starts.

**Listen for matchmaker events** The agents is listening for matchmaker events fired in the current room.

**Wait for propositions** The agent is waiting for propositions sent by a exploring matchmaker visiting the current room.

**Compare propositions with own dates** The agent compares the list of propositions with its own list of time periods.

**Send notification to owner** The agent sends notification to owner.

**Return matches** The agent returns matches.

**Send list of propositions** The agent sends list of propositions.

**Event: List of propositions** The agent waits for list of propositions.

**Event: List of matches** The agent waits for list of matches.

**Mode “Explore”**

**Search room** The agent searches room.

**Room found** The agent finds room.

**Matchmaker found** The agent finds matchmaker.

**No more rooms left** The agent has no more rooms left.

**Return to home room** The agent returns to home room.

**End** The agent ends.
**Return matches**  It returns the list of matching time periods back to the exploring matchmaker agent.

**Send notification to owner**  The agent sends a notification to the owner and thus makes the bridge from the virtual world to the real world. Currently, this is done by sending an email (shown in Figure 19), but it can be any kind of communication, like instant messaging, posting information to a website, sending a message to an **IRC**\textsuperscript{29} channel etc.

---

**Example of Matchmaker email**

```
From: johndoe@example.com
Subject: MatchMaker: new appointments while listening!
Date: Mon, 8 Nov 2004 18:53:07 +0100 (CET)

From: 2004-11-08 20:30
To: 2004-11-08 22:00
Explorer Information: 192.168.0.127/R11/m2r4
Explorer Description: I’d like to play chess.
Explorer Email: chesspro@example.com
Listener Information: 192.168.0.127/R11/m2r11
Listener Description: Any board game is ok.
Listener Email: johndoe@example.com

From: 2004-11-08 18:00
To: 2004-11-08 19:00
Explorer Information: 192.168.0.127/R11/m2r4
Explorer Description: I’d like to play chess.
Explorer Email: chesspro@example.com
Listener Information: 192.168.0.127/R11/m2r11
Listener Description: Any board game is ok.
Listener Email: johndoe@example.com
```

**Figure 19:** Example of email generated by a listening “Matchmaker” agent

4.3.2 **Mode “Explore”**

The right side of Figure 18 shows the flow diagram of the matchmaker in explorer mode:

**Start** The agent starts.

**Search room**  The agent is searching for the next room he has not visited yet.

**Room found** The agent finds a next room to visit and enters this room.

**Matchmaker found**  The agent finds a listening matchmaker agent in the current room.

\textsuperscript{29}Internet Relay Chat
Send list of propositions  It sends his list of date periods to the listening agent.

Wait for matches  The agent is waiting for matches being sent by the listening agent.

No more rooms left  There are no more rooms left to explore.

Return to home room  The agent returns to its home room and sends an email to its owner with all the arranged meetings (similar to Figure 19).

4.4  Further Examples

In this section, two other possibilities to use MaDViWorld in a different context are presented. While MaDViWorld itself is a framework dedicated to virtual worlds, its distributed architecture allows to think of applications relying solely on this fact. The following two applications were not implemented.

4.4.1  Distributed Audio Compression

Nowadays, digital audio data is very common. Certain environments like radio and TV stations save literally terabytes\textsuperscript{30} of data. One common problem is format conversion, be it from a lossless source (like WAVE files) to a lossy format (like MP3 files) or to another lossless, but compressed format (like FLAC files). This process is CPU intensive. When dealing with thousands of audio tracks, parallelism is easily attained and distributing the load scales well. This means distributing the work load over $n$ machines results in a speed up of almost factor $n$ compared with a single machine.

Figure 20 identifies these five types of agents:

- **cdda2wav\textsuperscript{31}**: While not really an agent, this application converts data from an audio CD to wave files and stores them on a file server.

- Directory: This agent serves as central yellow book. Every other type of agent in this scheme registers with it by providing its location, name and capabilities.

- Producer: A Producer agent is essentially a waiting queue of files ready to be converted (each with a specified target format). The cdda2wav application writes URI\textsuperscript{s} of new files into one of these producers’ queues.

- Consumer: A Consumer agent connects to a Producer and converts one file at a time from the waiting queue.

- Logger: To ensure correct operation, every action is logged with one of these agents.

\textsuperscript{30}1 terabyte equals 1000 gigabytes, i.e. $10^{12}$ bytes.

\textsuperscript{31}cdda2wav is used to extract digital audio from audio CDs\textsuperscript{14}. 
There can be more than one instance of each of these agent types. Normally, there is a Consumer for each machine, a producer for each `cdda2wav` instance (although more are possible), one or more Logger and one Directory instance.

![Diagram](image)

**Figure 20:** Using MaDViWorld to distribute load when converting audio data

### 4.4.2 Distributed Digital Still Images Decoding

Another source of massive amounts of digital data are digital still cameras. Image capturing sensor sizes grow bigger every 6 months. If two or three mega pixel cameras were the best money could buy two years ago, eight mega pixel sensors are standard as of 2004. Every one of these pixels need to be stored somewhere, thus file sizes increase proportionally. While six mega pixel sensors create **RAW** files of about 6 megabytes (average), eight mega pixel sensor files are typically around 8 megabytes in size.

Conversion of these files is CPU intensive. There are two areas, where conversion is needed. The first is when the photographer decides to shoot **RAW** files. These files need to be interpolated\(^{32}\) to common RGB files for the subsequent steps in the work flow. The second case is when existing files need to be resized, marked with watermarks or prepared for web display.

The conceived scheme resembles very much the architecture presented in the previous section. There are slight changes in conversion logic (different external codecs\(^ {33}\)):

---

\(^{32}\)This is also called de-mosaicing and is necessary due to the color filters found on image sensors.

\(^{33}\)Compression Decompression
• Source: This program provides the sources files and interacts with the directory and one or multiple Producers. Except for details, this program does not have to be adapted.

• Directory: This agent remains essentially the same as in the audio case.

• Producer: There is no difference in the Producer between the audio case and digital still images.

• Consumer: The Consumer has special codecs for conversion, but remains the same otherwise.

• Logger: The Logger does not have to be changed at all.
5 Changes to MaDViWorld

This Chapter covers which changes were necessary in order to make MaDViWorld support mobile agents. Each section covers a certain aspect of the framework.

5.1 Security and Agents in MaDViWorld

Until spring 2004, there was only a rather simple security layer available in the MaDViWorld framework. In essence, when creating an avatar, a type could be chosen (either Professor or Student). Upon the creation of a room, an attribute specifying which type of avatar has the right to enter the room had to be specified (this could be Professor, Student or Everybody). While certainly not very sophisticated, the scheme met the requirements.

Additionally, the implementation demanded that objects could only be removed from a room if this room was “owned” by the avatar requesting the removal. To support software agents, the implementation was changed so that each agent object had a special flag, which was checked when it tried to remove other objects. This is shown in the following code excerpt (Figure 21):

```java
public void removeObject(String avatarname, String objectname) {
    WObject object = (WObject) objects.get(objectname);
    if (getOwner().equals(avatarname) ||
        object.getType() == Constants.AGENT_OBJECT) {
        // remove object.
    } else {
        // don‘t remove object.
    }
}
```

Figure 21: Old security code excerpt

In spring 2004, Patrik Fuhrer implemented a new security layer. It is based on a challenge-response test. See Figure 22 for a code excerpt.

The new scheme consists of these three interfaces:

RoomAccessor This interface is responsible for making available a question and checking the answer. It consists of these methods:

- Question getQuestion() and
- Room checkAnswer(Object answer).

See [65] for more information.
Question This interface has only one method:

- Object execute(Solver s)

Solver This class has only one method:

- String getPassword().

The UML sequence diagram in Figure 23 shows the time flow:

1. getRoomReference() calls getRoomAccessorReference() and gets the accessor for the room.

2. getRoomReference() calls roomAccessor.getQuestion() and receives the question object for this room.

3. getRoomReference() locally calls question.execute(solver) with the user-provided solver object.

4. Question.execute() calls solver.getPassword() and returns the answer back to getRoomReference().

5. getRoomReference() calls roomAccessor.checkAnswer(answer) and receives a reference to an appropriate (according to the answer) proxy for the actual room.

```java
public Room getRoomReference(String name, String host, Solver solver) {
    try {
        RoomAccessor roomAccessor = getRoomAccessorReference(name, host);
        Question q = roomAccessor.getQuestion();
        Object answer = q.execute(solver);
        // Answer computed, will ask the secure roomproxy
        Room room = roomAccessor.checkAnswer(answer);
        // Obtained secure roomproxy, ready to return
        return room;
    }
    catch (RemoteException e) {
        // Error while getting a secure room proxy
        return null;
    }
}
```

Figure 22: Challenge-response code excerpt
Figure 23: Challenge Response scheme sequence diagram
5.2 Improved Code Mobility

When the work on agents in MaDViWorld began, the framework used a special scheme to achieve 
code mobility that was based on standard RMI\textsuperscript{35}. However, to overcome certain 
limitations of the RMI architecture, the scheme to allow objects to move from one host to another was rather complex (see Figure 24). Each Java class file had to be transferred individually 
from the old to the new host, which also resulted in considerable overhead.

![Figure 24: Old code mobility scheme](image)

When an object is placed, code transportation was done like the following:

1. The RoomWizard (on host 10.0.0.1) calls the RMI method setObject() on the remote host (10.0.0.2).

2. On the remote host, the RMI service (part of the JFC\textsuperscript{36}) gets all needed Java classes per HTTP from the local host’s HTTP Class Server.

3. The RoomWizard calls the RMI method getClass() on the remote host to initiate a re-get of all Java classes over HTTP (this time with the request GE2 instead of GET). This makes it possible to store the classes locally on the disk.

4. The framework exports the same object as RMI stub (with the codebase annotation pointing to its own HTTP class server, which serves the locally saved classes from the previous point).

\textsuperscript{35}Remote Method Invocation, the Java equivalent of RPC (Remote Procedure Call)
\textsuperscript{36}Java Foundation Classes, see [40].
In addition, the old scheme was conceived to support only code mobility between the following parties:

- **RoomWizard - RoomServer**: This is used for the first instantiation of an object (i.e., when placing an object into a room on a server).
- **Avatar - RoomServer**: Used when an avatar puts an object in to his bag and when the avatar puts an object from his bag back to a room.

The framework had to be adapted to allow RoomServer - RoomServer code mobility as well.

Problems arose when I developed an agent moving by itself (RoomServer - RoomServer mobility). Together with Patrik Fuhrer we decided to re-implement the code mobility part. The new mobility scheme is simpler, more elegant and less error-prone. In contrast to the old implementation, only one **JAR** file containing all Java Class files has to be transmitted when moving an object from one host to another.

When an object is placed in a remote room, code transportation is now done like the following (see Figure 25):

1. A client (this can be a **RoomServer**, a **RoomWizard** or the avatar application) on host 10.0.0.1 calls the RMI method **setObject()** on the remote host (10.0.0.2).
2. On the remote host, the method **Administration.getObjectClasses()** requests the **JAR** file containing all resources (Java classes, graphics, files etc.) from the local host. The resources are uncompressed to the local disk.
3. Still on the remote host, the RMI service (part of the **JFC**) gets all needed Java classes per **HTTP** from the local host’s **HTTP** Class Server.
4. The framework exports the same object as RMI stub (with the codebase annotation pointing to its own **HTTP** class server, which serves the locally saved classes originating from the **JAR** file).

Another problem that was that the client window froze after launching the Explorer agent and would only become responsive again after the agent was back in the originating room. The reason for this behavior was that the client ran in a single thread, which blocked when the Explorer moved to another room. We solved this problem by starting a new thread when for each object that was put in a room.

In early versions, the agents deduced from being called by the framework with the **setState()** method that they were new in the room and acted accordingly (inspect room etc.). However, the sequence of method calls were different after the implementation of the new security features (discussed in Chapter 5.1) only because the one of the two threads involved in the operation was slower. This resulted in the paradox situation that setting up the object terminated only after the agent started its work (by tapping the invocation of **setState()**). As a consequence, the internal state of the framework was inconsistent and prevented the successful operation of the agent (and the server in general).

[^37]: **Java Archive**, actually a renamed **ZIP** file.
To remedy this problem, a new method called `postInstall()` was introduced to the framework’s interface `core.WObject` and default implementation `wobject.WObjectImpl`. It is called automatically after the process of setting up the object in the room completes.

### 5.3 Porting to Linux

As MaDViWorld was developed and tested on Microsoft Windows, it did not run very well on Linux, even if it was completely written in Java\(^{38}\). I refactored the framework slightly in order to avoid hardcoded backslashes\(^{39}\), but instead rely on dynamic operating system recognition and independently of the platform currently running using the correct file separator. This was done by using the Java method `java.lang.System.getProperty("file.separator")`.

### 5.4 Console Version of XML Setup

The diploma thesis of Patrick Pauchard [50] contributed a visual world editor to MaDViWorld. Its main feature is the creation of a XML representation of a designed world. The environment can as well read in a valid XML file representing a MaDViWorld world and add the specified rooms, doors and objects.

The XML Setup application was improved by me to allow the creation of doors when deploying the XML file, which was not possible before. Together with Patrik Fuhrer, the XML DTD was extended so

---

\(^{38}\) In fact, it was more WOTA (Write Once - Test Anywhere) than the promised WORA (Write Once - Run Anywhere).

\(^{39}\) Denoting file separators in Microsoft Windows. Unix-like systems (Linux and MacOS X included) use a slash (`/`), MacOS used a colon (`:`) until version 9-
that doors to rooms on remote hosts were possible (even without those remote hosts being defined in the same XML file):

The old syntax is showed in Figure 26, the new syntax in Figure 27.

```xml
<doorlist>
  <door>
    <!-- door to a room on the same host as the current room -->
    <roomname>corridor</roomname>
  </door>
</doorlist>
```

**Figure 26: Old XML Syntax**

```xml
<doorlist>
  <door>
    <!-- door to a room on the host 'myDistantHost' -->
    <roomname>corridor</roomname>
    <host>myDistantHost</host>
  </door>
</doorlist>
```

**Figure 27: New XML Syntax**

To ease and speed up the testing cycle while developing agents, instead of having to perform endlessly the same GUI operations with the setup application, a simple client of the underlying application logic was created. This client reads in a given valid world XML file and adds the specified entities to a given MaDViiWorld server, without popping up a GUI. More importantly, this solutions allows testing of agents under the same, controlled conditions in each test cycle.

### 5.5 Versioning System

To improve developer efficiency, Patrik Fuhrer and I decided to implement a CVS repository for the MaDViiWorld project. On top of common CVS functionality it features a web front end for browsing the repository (implemented with ViewCVS [63]). Furthermore, each check-in (be it new, modified or removed files), along with the log entry entered by the developer is sent to a dedicated mailing list. This allows efficient team work, as developers don’t have to poll for information, but it is sent to them by email.

---

40 Concurrent Versions System, see [10] for more information.
41 For example by manually updating their local copy of the repository each day and scan for updates.
42 This is achieved by a customized and slightly modified commit-script called CVSreport [8].
5.6 Build System

When starting with this project, it became apparent that the then-used build system did not work well for me. It was based on Microsoft Windows batch files compiling the needed Java classes. Because I work with Linux, this would have meant to effectively support two versions of the scripts (one with batch files and one with shell scripts). We opted for Ant [3], a build system based on Java and XML offering a lot of flexibility through its modular architecture.

In the current version, the Ant scripts compile, package and deploy in one single step all needed Java classes and other resources.
6 Suitable Improvements

As mentioned at the beginning of this paper, MaDViWorld was not conceived as a platform for mobile software agents. Nevertheless, most requirements of such a platform are already met. The absence of other features however prevents its use as a full-fledged mobile software agent platform.

The first three Sections in this Chapter show features that do not conflict with the original philosophy of MaDViWorld being a framework for virtual worlds (Sections 6.1, 6.2 and 6.3). The remaining three Sections address issues only important for an agent platform and thus not necessary for MaDViWorld as a virtual world framework (Sections 6.4, 6.5 and 6.6). But with the exception of the centralized directories discussed in Section 6.4, all of them could be integrated into the framework without breaking the massively distributed requirements.

6.1 Dynamic Reloading

Should MaDViWorld be used to effectively support mobile agents in a production-like environment, the up time of the system is crucial. But because software is never finished, agents evolve over time and should replace older versions. The current framework however needs to be re-built and restarted every time a piece of an agent is changed. A solution to this problem could be to dynamically re-load the code of the changed agent. To prevent confusion between different versions of the same agent, an internal versioning scheme could be implemented.

6.2 Persistence

Another component that is currently missing is a stable, easy and efficient way to achieve persistence of the state of an object. The current solution with a hash table the framework saves before moving the object and re-injects after movement is not suited even for moderately complex agents like the Explorer agent (see section 4.2) or even more so the Matchmaker agent (discussed in section 4.3).

A flexible, simple to use and efficient to implement solution would be the inclusion of a middle-ware persistence mechanism. A relational persistence product like Hibernation [34] (in combination with an embedded Java SQL engine like HSQLDB [35]) or a JDO [43] implementation could be used.

A simpler approach could be to mandate a certain name convention for fields to mark them as stateful. The code example in Figure 28 shows how this could look like when using a leading underscore to mark fields of the implementing class as stateful. Before moving an agent, the MaDViWorld framework would check with runtime reflection which fields of the agent start with an underscore and save only these for restoration after movement. Compared with a single state field, this scheme would be a lot more flexible for the developer of an agent and result in better readable and maintainable code.

---

43 Also called introspection.
44 The process of saving would probably be implemented using serialization.
public class MyAgentImpl extends WObjectImpl implements MyAgent {
    private String _statefulField1;
    private Vector _statefulField2;
    private String statelessField1;
    private Vector statelessField2;

    public MyAgentImpl(){
        ...
    }
}

Figure 28: Using underscores to mark stateful fields

6.3 Programmable Timers

Frequently, it would be nice to have a mechanism that calls back an agent or another object after a certain time. Such programmable timers would allow objects to manage situations where busy waiting is not desired (performance-wise) or possible. An exploring “Matchmaker” agent trying to communicate with a listening “Matchmaker” is fully dependent on the answer of the latter agent. As long as the answer is not received, the exploring agent cannot continue, as it is not run by the framework until it receives an event. Thus, if the exploring agent can program a timer in the current room to call him back after 10 seconds (so it can act if no event was received up to then), it can avoid a dead-lock situation and still continue its travel and successfully report back to its avatar.

public interface WContainer extends Remote, RemoteEventProducerRemote, RemoteEventProducerTemporal {
    ...
    public void registerTimer(RemoteEvent evt, int seconds);
    ...
}

Figure 29: Timer API proposal

The proposed API for this feature could look like shown in Figure 29. It extends the interface core.WContainer by the method registerTimer(RemoteEvent evt, int seconds). The usage pattern would look like this: An object prepares a RemoteEvent and sets the timer by calling registerTimer(evt, seconds). After the specified time, the framework would send out the buffered remote event and thus notify all registered listeners.
6.4 Centralized Directory

Certain problems can be solved with multiple agents collaborating. Others rely on open and casual collaboration, where different types of agents offer different services. Therefore, a centralized directory (Yellow Pages) should be made available, where all available agents register\(^{45}\). This is important to efficiently distribute work or search for idle agents of a certain type that can accept work or perform other tasks.

To further abstract agents from their physical location, a special kind of Agent Name Servers (ANS) should be conceived. Traditionally, they store globally unique tuples denoting agents in the form \((\text{hostname}, \text{port})\)^{46}. This can be limiting, since (mobile) agents potentially move and thus change their address. Other agents trying to communicate with such an agent will fail or - even worse - reach the wrong agent (which also moved and coincidentally received the same address). To prevent such inconsistencies, a lease scheme\(^{47}\) should be employed, assuring that each agent periodically renews its entry in the ANS and keeps it up-to-date.

6.5 Communication

MaDViWorld supports the notion of event within a limited range (typically the room the emitting object is in). This can be compared to a restricted broadcast, as every object in the same room can react to it if they want. Agents however need to cross the borders of the current room and in addition have to be able to perform peer to peer (unicast) communication\(^{48}\). Additionally, true broadcast communication to reach all agents in the world could be useful, although scalability would be an issue. Message Routers in combination with a Centralized Directory should be employed instead.

Third, asynchronous communication is important. It should be implemented either in the form of one (or multiple) failure tolerant Agent Message Routers (so agents do not have to be online necessarily when sending messages to them). Or, alternatively this functionality could be included in the facilitator agents discussed below. To communicate successfully, both parties need to be online, the sender is blocked until the receiving agent completed processing the message. Agent Message Routers operate by the store and forward\(^{49}\) paradigm and thus work around these limitations.

Another requisite is the ability to send and receive messages to and from other agents. Currently, the framework allows to implicitly send messages by firing events. But because the propagation of events is limited to the virtual room it was sent out\(^{50}\), agents relying on this mechanism won’t be able to communicate with agents in other virtual rooms.

\(^{45}\)Because this can be a single point of failure, some kind of election algorithm should be used to ensure there is always a working directory.

\(^{46}\)See also [22].

\(^{47}\)Similar to the Lease Renewal Service implemented in Jini [46].

\(^{48}\)Peer to peer communication is already possible with the current implementation if both peers are located in the same room.

\(^{49}\)See also [67].

\(^{50}\)See [26] for a formal discussion.
One solution to this problem could be facilitator agents\textsuperscript{51} that reside in a room with other agents\textsuperscript{52}. If any of the agents wishes to send a message to an agent in another room, it sends the message to the facilitator instead. The facilitator routes the message to the facilitator of the message’s target agent.

\textbf{Figure 30:} Federated system with two federations

Figure 30 shows how communication flows would look like:

1. Local messaging: \textbf{Agent 3} sends a message to \textbf{Agent 2}. No facilitator intervenes, because both agents are in the same room (here Room 1).

2. Remote messaging: \textbf{Agent 1} wants to send a message to \textbf{Agent 4} in Room 2. It sends the message to its local facilitator (in this case \textbf{Facilitator 1}). The facilitator sends the message to the addressee’s own local facilitator, \textbf{Facilitator 2}. This facilitator finally sends the message to \textbf{Agent 4}\textsuperscript{53}.

This scheme could be improved by making it opaque: The facilitator listens to all messages sent out by agents in the same room and routes it in a completely transparent fashion to the target facilitator if the receiver is not local to the room.

Instead of relying on special agent in each room, another solution could be put in place by implementing a wrapper around the current event sending and receiving API\textsuperscript{54}. This wrapper would treat agent messages differently and not restrict them to the local room.

\subsection*{6.6 Agent Communication Language}

Efficient communication is based on the condition that all concerned parties know how to understand a message. Because there can be a variety of different agents in a virtual world, each communication must

\textsuperscript{51}In the style of [30].

\textsuperscript{52}One facilitator per room.

\textsuperscript{53}This can be done in different ways, one could be to let a specialized message agent move to the addressee.

\textsuperscript{54}Application Program Interface
be preceded by a fixation of the used ontology, the shared semantics, because the meaning of a message depends on which ontology is used.

Agent Communication Languages (ACL) provide these possibilities (and more) and should therefore be envisioned for an inclusion in MaDViWorld.

The current eventing API is very open and consists of the elements shown in Figure 31.

```java
1  public RemoteEvent(java.lang.Object source,
2       long eventID,
3       long seqNum,
4       java.lang.String des)
```

```java
1  public void addAttribute(java.lang.Object key,
2       java.lang.Object value)
3  public java.lang.Object getAttribute(java.lang.Object key)
```

**Figure 31: Eventing API**

The API is cumbersome to use, even more when dealing with agents that often broadcast messages. The container used to talk to the API is a simple hash table and is very generic. While this provides maximal flexibility, complex communication systems can be hard to develop and maintain.

To be an effective communication API, certain constraints on coherence and completeness of messages should be put in place. First, a clear declaration of the source and target name, location and type of agent seems needed. Because different type of agents use different ontologies\(^{55}\), the used ontology should be made mandatory.

Standards like KQML\(^{56}\) or (preferably) FIPA ACL should be added on top of the currently available communication API.

\(^{55}\)The conceptualization of the world (objects, qualities, distinctions and relationships) [16]. See also [33].

\(^{56}\)Knowledge Query and Manipulation Language
7 Conclusion

Compared to other software agent toolkits or frameworks, our changes to \textbf{MaDViWorld} resulted in a framework that certainly can be described as software agent platform. It includes the most important features like autonomy, code mobility, intelligence, communication. However, it is not a platform that focuses on software agents - neither in the present form nor historically. But even if development of agents is not as easy and efficient as with dedicated platforms, three interesting agents were implemented.

The first agent, called “Hopper”, serves as a simple proof-of-concept of mobility. The second agent, called “Explorer”, wanders autonomously around a virtual world and draws a graphical map of its findings. The third, called “Matchmaker”, tries to arrange appointments on behalf of humans with other agents of its type.

The biggest challenge of this work was to master the complexity of the framework. Because my work used it in such a fundamental different way than any other developer before, a lot of (until then undiscovered) bugs and design inadequacies surfaced and had to be dealt with, either by fixing (where feasible) or working around them (when fixing them was not possible).

Improvements of the framework were made mainly in the following areas:

- code mobility;
- security;
- build system, code repository, Linux compatibility.

Another challenge was the fact that distributed systems are very hard to debug, and \textbf{MaDViWorld} is no exception to this rule. Error tracing and problem solving in such a dynamic environment are difficult. A lot of time was invested in managing these two aspects.

Concluding it seems that implementing mobile software agents in \textbf{MaDViWorld} was certainly possible. However, the time spent mastering the framework complexity, following the development process of the framework, hunting down bugs and dealing with design issues resulted in an immense effort.

Concluding, the project:

- proved the feasibility of implementing and using mobile software agent technology to perform useful tasks within \textbf{MaDViWorld};
- helped improve the framework in order to make it a better platform supporting not only reactive but also proactive objects;
- showed some inherent limits of the framework as far as being a full fledged agent platform;
- revealed a small number of weaknesses in the framework during the implementation phase.
Indeed, it will be possible to further enhance the **MaDViWorld** framework by building a better communication infrastructure; by adding additional services like central directories, programmable timers and dynamic reloading; by improving persistence, as to better support mobile agents. In fact, except the latter, none of these changes go against the main philosophy of **MaDViWorld**.

Personally, working with an academic research project was a very enriching experience.
A Using Ant to Build and Run MaDViWorld

The provided Ant scripts allow to compile, package and deploy the MaDViWorld framework easily. First of all, the framework needs to be compiled. To do this, change to the MaDViWorld root directory, where the build.xml file resides and run the following command:

```
Ant scripts allow to compile, package and deploy the MaDViWorld framework easily. First of all, the framework needs to be compiled. To do this, change to the MaDViWorld root directory, where the build.xml file resides and run the following command:

```

```

```

```

```

```

```

Now you need to start the server component. This is best done in a window of its own, as it needs to be running the whole time you want to use MaDViWorld:

```

```

```

```

```

```

The third step consists of starting the setup application, so you can add some rooms and objects to the virtual world:

```

```

```

```

```

```

```

```

Finally, you want to start the client application, so you can actually log into the virtual world:

```

```

```

```

```

```

```

```
B MaDViWorld and Firewalls

This section concentrates on problems that may arise during the use of MaDViWorld on a computer that is protected with a firewall. The framework can use two different middlewares to achieve distributed computing: Jini or RMI. During the work on this paper, RMI was used as Jini would not work together with MaDViWorld on Linux (my main development platform) for unknown reasons.57

The RMI protocol uses direct socket connections over TCP. The range of observed TCP ports is $30000 - 65535$ (both in- and outbound)58.

The program rmid (RMI daemon) uses the fixed TCP port 1098, rmiregistry (used for registering RMI-capable classes) uses port 1099.59

The framework uses various HTTP class servers. They listen on ports 8083, 8084, 8085.

Here is an example of a firewall configuration, Shorewall [12]:

<table>
<thead>
<tr>
<th># ACTION</th>
<th>SOURCE</th>
<th>DEST</th>
<th>PROTO</th>
<th>DEST PORT(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT</td>
<td>fw</td>
<td>loc</td>
<td>tcp</td>
<td>8083</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>fw</td>
<td>loc</td>
<td>tcp</td>
<td>8084</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>fw</td>
<td>loc</td>
<td>tcp</td>
<td>8085</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>loc</td>
<td>fw</td>
<td>tcp</td>
<td>8083</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>loc</td>
<td>fw</td>
<td>tcp</td>
<td>8084</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>loc</td>
<td>fw</td>
<td>tcp</td>
<td>8085</td>
</tr>
<tr>
<td># rmid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCEPT</td>
<td>fw</td>
<td>loc</td>
<td>tcp</td>
<td>1098</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>loc</td>
<td>fw</td>
<td>tcp</td>
<td>1098</td>
</tr>
<tr>
<td># rmiregistry</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCEPT</td>
<td>fw</td>
<td>loc</td>
<td>tcp</td>
<td>1099</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>loc</td>
<td>fw</td>
<td>tcp</td>
<td>1099</td>
</tr>
<tr>
<td># rmi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCEPT</td>
<td>loc</td>
<td>fw</td>
<td>tcp</td>
<td>$30000:65535$</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>fw</td>
<td>loc</td>
<td>tcp</td>
<td>$30000:65535$</td>
</tr>
</tbody>
</table>

57 Considerable effort was put in solving this problem, but abandoned in favor for more specific, mobile agent development.
58 The given port range was observed by analyzing firewall logs and using the network sniffer Ethereal [9]. This was necessary as the used RMI implementation from Sun is not open-source, and could not be analyzed because of this.
59 These two ports are the default values and can differ depending on the configuration. MaDViWorld uses the default ports.
C Linux and RMI

Sun’s implementation of RMI shows some strangeness on Linux. The first problem arises because Linux kernels up to (and including) version 2.4 count threads as processes. Since RMI applications create hundreds of threads, this might use up all your allowed processes defined with `ulimit` and thus prevent a successful use of MaDViWorld. To fix this, run the following command before you start the MaDViWorld framework:

```
ulimit -u 4096
```

The second problem lies in the way RMI looks up addresses. If a RMI call from a Linux machine is done, a reply address is transmitted over the connection. Unfortunately, the implementation takes the IP-address from the first line in `/etc/hosts`. Normally, the first line happens to be the entry pointing from localhost to the loop-back device (IP address 127.0.0.1).

So, if your `/etc/hosts` looks like the following, RMI will assume its own (external) IP address is 127.0.0.1, which is clearly wrong.

```
127.0.0.1 localhost
```

As a remedy, you can change your `/etc/hosts` file to look like the following (given you do not use DHCP and your static, real IP address is 192.168.0.42 and your machine is called mybox, and it is in the mydomain domain):

```
192.168.0.42 mybox.mydomain
127.0.0.1 localhost
```

Another, cleaner solution would be to set the `java.rmi.server.hostname` property of the RMI server:

```
java -Djava.rmi.server.hostname=mybox ...
```

Or, if you are using the supplied MaDViWorld Ant scripts:

```
<jvmarg value="-Djava.rmi.server.hostname=mybox"/>
```

---

60 This sets the maximal number of processes allowed to run simultaneously by the current user. The standard configuration in combination with Kernel ≤ 2.4 will most likely not suffice in terms of allowed processes.

61 This behavior is described as well in Sun’s Bug database [60].
Alternatively, if you want the highest possible flexibility (if you are using DHCP for example) and do not want to hard code anything, you can set the `java.rmi.server.useLocalHostname` property to `true`.

```
java -Djava.rmi.server.useLocalHostname=true ...
```

Or, if you are using the supplied MadViWorld Ant scripts:

```
<jvmarg value="-Djava.rmi.server.useLocalHostname=true"/>
```

---

62This will use the first entry in `/etc/hosts` containing a dot (.) character and is described at [59].
D Testing on Multiple Machines

Testing of a distributed system can be quite challenging for a number of reasons. One problem is the distribution of the framework code to all participating machines. Particularly during development, the code is evolving quickly and test cycles can be as short as minutes. This fact calls for an efficient way to synchronize all machines. rsync [61] was developed by the same folks that wrote Samba [51]. It uses a differential algorithm to quickly synchronize two distant (or local) directories and works preferably over SSH (Secure Shell).

The following Bash script synchronizes the current local directory with two other machines on the network:

```bash
#!/bin/bash
rsync -a --exclude="deploybin/server/build.properties" \ 
--delete . nicola@mybox1:~/MaDViWorld
rsync -a --exclude="deploybin/server/build.properties" \ 
--delete . nicola@mybox2:~/MaDViWorld
```
E  Source Code and Related Material

The accompanying CD-ROM contains all the source code of the framework, including the three agents presented in this paper. The latex sources (including all images) and a compiled PDF version of this document can be found at doc/pdf/.

The contents of the CD-ROM can also be downloaded from these two locations:
http://diuf.unifr.ch/softeng/student-projects/completed/fankhauser
http://variant.ch/phpwiki/MobileAgentsInMaDViWorld

Structure of directories

build/
deploybin/
  client/
  server/
  setup/
doc/
  javadoc/
  latex/
    images/
  pdf/
lib/
misc/
source/
F Used Software

- Eclipse 2.1, 3.0.1 (Java IDE)
- JEdit 4.2pre8 (text editor)
- Ant 1.6.1 (build tool)
- Java SDK 1.4.2
- Gimp 2.0.1 (bitmap graphics manipulations)
- OpenOffice 1.1.1 (vector graphics manipulations)
- pdfLatex 3.14159
- Debian 3.0, Linux 2.4.22
- Ethereal 0.10.0 (network sniffer)
- Together 6.0 (UML tool)
G License

Copyright (c) 2004 Nicola David Fankhauser

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
References


   http://ant.apache.org/ (last visited: 2004-10-09)


[6] Borland Together Control Center  
   http://www.borland.com/together/ (last visited: 2004-10-09)


[8] V. Caron, 2001: *CVSreport*  
   http://home.gna.org/cvsreport/ (last visited: 2004-10-09)

   http://www.ethereal.com/ (last visited: 2004-10-09)

[10] CVS - Concurrent Versions System  
    http://www.cvshome.org/ (last visited: 2004-10-09)


    http://shorewall.sourceforge.net/ (last visited: 2004-10-09)

    http://eclipse.org/ (last visited: 2004-10-09)

[14] H. Eißfeldt, 2000: *cdda2wav - the sampling utility*  
    http://www.cdda2wav.de/ (last visited: 2004-10-09)
http://www.kr.tuwien.ac.at/research/reports/rr0102.ps.gz (last visited: 2004-10-09)


[17] FLAC - Free Lossless Audio Codec
http://flac.sourceforge.net/ (last visited: 2004-10-09)

[18] The Foundation for Intelligent Physical Agents: *FIPA*
http://www.fipa.org/ (last visited: 2004-10-09)

http://www.refactoring.com/ (last visited: 2004-10-09)

http://www.msci.memphis.edu/~franklin/AgentProg.html (last visited: 2004-10-09)


http://www.race.u-tokyo.ac.jp/~glenn/javaprog/project/JavaAgent/documentation/addressing.html (last visited: 2004-10-09)

[23] P. Fuhrer, 2003: *MaDViWorld (Massively Distributed Virtual Worlds)*, University of Fribourg, Switzerland
http://diuf.unifr.ch/softeng/projects/madviworld/ (last visited: 2004-10-09)

http://diuf.unifr.ch/people/fuhrer/publications/internal/madviworld.pdf (last visited: 2004-10-09)

http://diuf.unifr.ch/people/fuhrer/studproj/invernizzi/metalpanic.html (last visited: 2004-10-09)

[38] JACOMMA - Java Communicating Agents
http://jacoma.sourceforge.net/ (last visited: 2004-10-09)

[39] Java Agent DEvelopment Framework
http://jade.tilab.com/ (last visited: 2004-10-09)

[40] Java Foundation Classes (JFC/Swing)

[41] Java Programming Language
http://java.sun.com/ (last visited: 2004-10-09)

[42] Javadoc Tool
http://java.sun.com/j2se/javadoc/ (last visited: 2004-10-09)

[43] Java Data Objects
http://access1.sun.com/jdo/ (last visited: 2004-10-09)


[45] JiVE - JAFMAS integrated Visual Environment
http://www.ececs.uc.edu/~abaker/JiVE/ (last visited: 2004-10-09)


[47] JiVE - JAFMAS integrated Visual Environment
http://www.ececs.uc.edu/~abaker/JiVE/ (last visited: 2004-10-09)

http://www.jython.org/ (last visited: 2004-10-09)


http://diuf.unifr.ch/people/fuhrer/studproj/pauchard/xmlmadvivworld.html (last visited: 2004-10-09)
[51] Samba Team, 1998-2004: Samba
   http://www.samba.org/  (last visited: 2004-10-09)

[52] M. Schmehl, 2003: GraphOpt
   http://graphopt.sourceforge.net/  (last visited: 2004-10-09)


[56] SpamAssassin
   http://www.spamassassin.org/  (last visited: 2004-10-09)

[57] Sun Microsystems, 2002: Core J2EE Pattern Catalog Core J2EE Patterns - Transfer Object
   http://java.sun.com/blueprints/corej2epatterns/Patterns/TransferObject.html  (last visited: 2004-10-09)

   http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmi-properties2.html  (last visited: 2004-10-09)

   http://java.sun.com/j2se/1.4.2/docs/guide/rmi/faq.html%23netmultihomed (last visited: 2004-10-09)
   http://java.sun.com/j2se/1.4.2/docs/guide/rmi/faq.html%23nethostname  (last visited: 2004-10-09)

[60] Sun Microsystems Bug Database: java.net.InetAddress.getLocalHost() returns 127.0.0.1

   http://rsync.samba.org/  (last visited: 2004-10-09)

   http://abiody.com/jfipa/architecture/  (last visited: 2004-10-09)

[63] ViewCVS - Viewing CVS Repositories
   http://viewcvs.sourceforge.net/  (last visited: 2004-10-09)
   http://citeseer.nj.nec.com/wooldridge00agentoriented.html (last visited: 2004-10-09)


[67] Wikipedia: Store and Forward  